Dialogic
Making Innovation Thrive™

Dialogic® Vision™ CCXML Developer's Manual

June 2009 64-0409-01

www.dialogic.com

Copyright and legal notice

Copyright © 2008-2009 Dialogic Corporation. All Rights Reserved. You may not reproduce this document in whole or in part without permission in writing from Dialogic Corporation at the address provided below.

All contents of this document are furnished for informational use only and are subject to change without notice and do not represent a commitment on the part of Dialogic Corporation or its subsidiaries ("Dialogic"). Reasonable effort is made to ensure the accuracy of the information contained in the document. However, Dialogic does not warrant the accuracy of this information and cannot accept responsibility for errors, inaccuracies or omissions that may be contained in this document.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH DIALOGIC® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND DIALOGIC, DIALOGIC ASSUMES NO LIABILITY WHATSOEVER, AND DIALOGIC DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF DIALOGIC PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Dialogic products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Due to differing national regulations and approval requirements, certain Dialogic products may be suitable for use only in specific countries, and thus may not function properly in other countries. You are responsible for ensuring that your use of such products occurs only in the countries where such use is suitable. For information on specific products, contact Dialogic Corporation at the address indicated below or on the web at www.dialogic.com.

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic and no such licenses are provided except pursuant to a signed agreement with Dialogic. More detailed information about such intellectual property is available from Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. **Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.**

Any use case(s) shown and/or described herein represent one or more examples of the various ways, scenarios or environments in which Dialogic® products can be used. Such use case(s) are non-limiting and do not represent recommendations of Dialogic as to whether or how to use Dialogic products.

Dialogic, Dialogic Pro, Brooktrout, Diva, Cantata, SnowShore, Eicon, Eicon Networks, NMS Communications, NMS (stylized), Eiconcard, SIPcontrol, Diva ISDN, TruFax, Exnet, EXS, SwitchKit, N20, Making Innovation Thrive, Connecting to Growth, Video is the New Voice, Fusion, Vision, PacketMedia, NaturalAccess, NaturalCallControl, NaturalConference, NaturalFax and Shiva, among others as well as related logos, are either registered trademarks or trademarks of Dialogic Corporation or its subsidiaries. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement.

The names of actual companies and product mentioned herein are the trademarks of their respective owners.

This document discusses one or more open source products, systems and/or releases. Dialogic is not responsible for your decision to use open source in connection with Dialogic products (including without limitation those referred to herein), nor is Dialogic responsible for any present or future effects such usage might have, including without limitation effects on your products, your business, or your intellectual property rights.

Revision history

P/N 64-0409-01

Revision	Release date	Notes
A	June 2009	DEH/BK, Dialogic® Vision™ CX Video Gateway 4.1 and Dialogic® Vision™ VX Integrated Media Platform 4.1
Last modified: June 8, 2009		

Refer to www.dialogic.com for product updates and for information about support policies, warranty information, and service offerings.

Table Of Contents

Chapter 1: Introduction 5

Chapter 2: The CCXML interface 7
Overview of the CCXML interface 7
Supported transfer scenarios 7
Configuration 8
Limitations 8
Supported CCXML elements 8
Supported CCXML session variables 11

Chapter 3: Using the CCXML interface 13
Starting a new CCXML session 13
Exchanging data between CCXML and VoiceXML 14
CCXML event processing 14
Additional attributes in the connection.alerting event 15
substate attribute in the connection.signal event 15
Reason mapping in the connection.failed event 15
CCXML logging 15
General CCXML scenarios 15
Multiple party audio conferencing 16
Media splitting for eavesdropping calls 17
Third party call control 18
Outbound SIP calls with preferred audio codecs 19
CCXML application definition file 19

Chapter 4: CCXML sample applications 23
Overview of CCXML sample applications 23
Configuring the sample applications 23
Calculating resources 24
Call recording 24
Resources and capacity 25
Running the application 26
Application logic 27
Music on hold 27
Resources and capacity 28
Running the application 29
Application logic 30
Music on hold with a wait queue 31
Resources and capacity 32
Running the application 33
Application logic 34
Third party call with transfer out 36
Resources and capacity 37
Running the application 38
Application logic 39
Whisper coaching 45
Resources and capacity 46
Running the application 47
Application logic 47

Chapter 5: Gateway routing table application 49
Overview of the Gateway routing table application 49
Object model 49
Call states 50
Routing a direct call 51
Routing a transferred call 53
Disconnect and error handling 55

Chapter 6: Glossary 59

1 Introduction

The *Dialogic® Vision™ CCXML Developer's Manual* describes how to use the CCXML interface to configure and develop CCXML applications for the Dialogic® Vision™ CX Video Gateway and the Dialogic® Vision™ VX Integrated Media Platform. The CCXML interface is part of the Dialogic® Vision™ Call Server, which comes with the Vision™ CX Video Gateway and with all Vision™ VX Integrated Media Platform models except IP-only audio.

This manual assumes that you are familiar with the CCXML language and coding web applications. This manual also assumes that you have read the *Dialogic® Vision™ CX Video Gateway Administration Manual* or *Dialogic® Vision™ VX Integrated Media Platform User's Manual*. The term Vision™ Server is used in this manual to refer to both the CX Video Gateway and the VX Integrated Media Platform collectively.

Note: The products to which this document pertains are part of the NMS Communications Platforms business that was sold by NMS Communications Corporation ("NMS") to Dialogic Corporation ("Dialogic") on December 8, 2008. Accordingly, certain terminology relating to the products has been changed. Below is a table indicating terminology that was formerly associated with the products, as well as the new terminology by which the products are now known.

Former terminology	Current terminology
Vision CX Gateway	Dialogic® Vision™ CX Video Gateway
Vision VoiceXML Server	Dialogic® Vision™ VX Integrated Media Platform

2 The CCXML interface

Overview of the CCXML interface

CCXML (Call Control Extensible Markup Language) is a web-based language that provides call control functionality. Use CCXML to write applications that provide call control for the duration of a call. Applications can include:

- Accepting or rejecting an incoming call.
- Routing a call.
- Answering an incoming call and connecting it to a VoiceXML dialog that plays a script.
- Placing an outbound call.
- Disconnecting a call.
- Transferring a call.
- Creating a conference for multiple calls.

The CCXML version implemented in the Dialogic® Vision™ Call Server is based upon the W3C Working Draft of CCXML dated 29 June 2005. For more information, see http://www.w3.org/TR/2005/WD-ccxml-20050629.

The CCXML interface to VoiceXML dialog services is based upon the document *SIP Interface to VoiceXML Media Services* dated 18 November 2006. For more information, see http://www.ietf.org/internet-drafts/draft-burke-vxml-03.txt.

CCXML functions with ISDN, ISUP, SIP, and video (3G-324M and IP) calls. Standard CCXML has no special features for video calling.

Supported transfer scenarios

The Vision™ Call Server supports the use of CCXML to perform bridge and blind transfers. These transfer types are available for audio-to-audio, video-to-video, and video-to-audio data transfers within the following protocols:

Incoming protocol	Outgoing protocol
IP	IP
	ISDN
	ISUP
ISDN	IP
	ISDN
	ISUP
ISUP	IP
	ISDN
	ISUP

To perform a blind transfer, use the CCXML <redirect> element. To perform a bridge transfer, use the <createcall> element to create a connection to the destination, and then join the source and destination legs together.

Configuration

CCXML settings for file names, file paths, logging levels, and so forth are pre-configured in the *callserver.conf* file in the *vx/callserver/conf* directory, where *vx* is the default installation directory, */opt/nms/vx*. The defaults are appropriate for most sites. You can modify many of these defaults, using the Vision™ Console. For information, see the *Dialogic® Vision™ CX Video Gateway Administration Manual* or the *Dialogic® Vision™ VX Integrated Media Platform User's Manual.* You can modify all defaults by editing the *callserver.conf* file directly, as described in the *Dialogic® Vision™ Call Server Administration Manual*.

Limitations

The Vision Call Server does not support the following CCXML elements:

- <merge>
- <meta>
- <metadata>

Supported CCXML elements

The following table lists the CCXML elements that the Vision™ Call Server supports, and notes any differences from the W3C Working Draft of CCXML dated 29 June 2005. For more information about the CCXML standard, see http://www.w3.org/TR/2005/WD-ccxml-20050629.

Supported CCXML element	Notes
<accept>	The hints attribute supports the answertype sub-attribute, which determines what action to take in response to an incoming SIP call. This attribute is valid for all Vision™ Server PSTN models. Valid values for answertype: • ACCEPT - Sends an address complete message (ACM) to accept the call. • ANSWER - Sends an answer message (ANM) to answer the call. • CONNECT - Sends a connect message (CON) to connect the call. • FULLANSWER - Send both an ACM and an ANM to accept and then answer the call. Default: FULLANSWER
<assign>	No differences from the standard.
<cancel>	No differences from the standard.
<ccxml>	No differences from the standard.
<createcall>	The aai attribute is supported for outgoing SIP connections only. The timeout attribute is supported for PSTN calls only. The hints attribute supports the following sub-attributes: • addressnature - Address nature indicator of the called party.

Supported CCXML element	Notes
	• audiocodec - Requests one of the following codecs: amr, g723, g726, g729, pcma, or pcmu. • mode - Call mode (voice or video). • route - Telecom route for the outgoing call. • tpcc - Requests third party call control. • delayack - Indicates whether to delay sending a SIP acknowledgement message (ACK) when generating an outbound SIP call. A value of true delays sending the ACK until after the call is joined to another leg with the <join> tag. This improves the quality of outbound video calls. A value of false (default) sends the ACK at the same time the call is joined to another leg. For example: `hints="{mode:'m',route:'r',addressnature:'a', 'delayack:true'}"`
<createccxml>	No differences from the standard.
<createconference>	No differences from the standard.
<destroyconference>	No differences from the standard.
<dialogprepare>	The following attributes are not supported: • conferenceid • maxage • maxstale • method The enctype attribute always requests a VoiceXML Interpreter instance. The metadirection attribute is always set to **both.** Media always flows in both directions. The dialog.user event is not supported. Use the namelist attribute to transfer data to the VoiceXML session.connection.ccxml.namelist variables in a VoiceXML application.
<dialogstart>	The following attributes are not supported: • conferenceid • maxage • maxstale • method The enctype attribute always requests a VoiceXML Interpreter instance. The metadirection attribute is always set to **both**. Media always flows in both directions. The dialog.user event is not supported. Use the namelist attribute to transfer data to the VoiceXML session.connection.ccxml.namelist variables in a VoiceXML application.
<dialogterminate>	No differences from the standard.

Supported CCXML element	Notes
<disconnect>	The following attributes are not supported: • hints • reason
<else>	No differences from the standard.
<elseif>	No differences from the standard.
<eventprocessor>	No differences from the standard.
<exit>	If a dialog exists, the variables contained in the namelist attribute are sent to the dialog in the SIP BYE message.
<fetch>	No differences from the standard.
<goto>	No differences from the standard.
<if>	No differences from the standard.
<join>	Joins inbound or outbound connections within a CCXML session. To join to a connection from another CCXML session, use <move> to put the connections in the same session before using <join>. Joining a video connection to an audio connection is allowed. When joining two SIP legs together, use the refcodec hint to indicate which leg's codec should be used. Set the hints attribute to {refcodec:'id1'} to refer to the codec of the first connection ID. If the other end point has a different audio or video codec, the Vision Call Server sends a RE-INVITE to negotiate the codec of the reference end point. Values for refcodec are: amr, g723,g726, g729, pcma, and pcmu. Set the hints attribute to mediasplit:'true' to activate media splitting. For information, see *Media splitting for eavesdropping calls* on page 17. For example: `hints="{refcodec:'amr' mediasplit:'true'}"`
<log>	No differences from the standard.
<move>	No differences from the standard.
<redirect>	The reason attribute is not supported. The hints attribute uses the following sub-attributes: • addressnature - Address nature indicator of the called party. • connecttimeout - Connection time out in ms (for PSTN calls only). • mode - Call mode (voice or video). • route - Telecom route for the outgoing call. For example: `hints="{mode:'m',route:'r',addressnature:'a'}"`
<reject>	The following attributes are not supported: • hints • reason
<script>	No differences from the standard.

Supported CCXML element	Notes
<send>	No differences from the standard.
<transition>	No differences from the standard.
<unjoin>	No differences from the standard.
<var>	No differences from the standard.

Supported CCXML session variables

The Vision™ Server supports the following CCXML session variables, as defined in the W3C Working Draft of CCXML dated 29 June 2005:

- session.conferences
- session.connections
- session.dialogs
- session.external
- session.id
- session.ioprocessors
- session.parentid
- session.startupmode
- session.uri

For more information, see http://www.w3.org/TR/2005/WD-ccxml-20050629.

3 Using the CCXML interface

Starting a new CCXML session

New CCXML sessions are triggered by the mechanisms shown in the following table:

Trigger	Action
Inbound call	When the Vision™ Server receives a call, it checks if the called number matches the number range for an entry on the CCXML application configuration page of the Vision™ Console. If it finds a match, the corresponding CCXML application is executed. If no CCXML application list is defined or if there is no match, the: • CX Video Gateway executes the CCXML application referenced by the CcxmlGatewayURI setting in the *callserver.conf* file (*gateway.ccxml* by default). • VX Integrated Media Platform executes the CCXML application referenced by the CcxmlInboundUri setting in the *callserver.conf* file (*inbound.ccxml* by default). For information about defining CCXML applications using the Vision™ Console, see the *Dialogic® Vision™ CX Video Gateway Administration Manual* or the *Dialogic® Vision™ VX Integrated Media Platform User's Manual*. For information about defining CCXML applications directly in the application definition file, see *CCXML application definition file* on page 19.
SIP INVITE from an application server	When the Vision™ Server receives a SIP INVITE from an application server, it executes the CCXML application specified by the NETANN URI. An example of a NETANN URI is: `application@callserver-host;ccxml=http://example.com/appli.ccxml;maxage=0;maxstale=0;aai=1234` where: • ccxml specifies the CCXML application to execute. • maxage specifies the max-age value of the cache-control header used when fetching the CCXML application on the remote HTTP server. • maxstale specifies the max-stale value of the cache-control header used when fetching the CCXML application on the remote HTTP server. • aai specifies an application-to-application information message.
Outbound call made from the CallPlacer interface	(Supported in the VX Integrated Media Platform only) When the VX Integrated Media Platform generates a call through the CallPlacer interface, it looks for an entry on the CCXML application configuration page of the Vision™ Console that has 9999 as the number range value. If it finds a 9999 entry, the corresponding CCXML application is executed, and the call is routed to the route specified by the <servicenumber> element in the placecall request. If no CCXML application list is defined or if there is no match, the VX Integrated Media Platform executes the CCXML application referenced by the CcxmlOutboundUri setting in the *callserver.conf* file (*outbound.ccxml* by default). For more information about the CallPlacer interface, see the *Dialogic® Vision™ VX Integrated Media Platform User's Manual*.

Trigger	Action
<createccxml> element	Specifying the <createccxml> element from within a CCXML application opens a new CCXML session. The new session operates independently of the existing session, and it has a separate variable space for the CCXML documents that it executes.

createsession event I/O processor	When the Vision™ Server receives a request of the following form, it creates a new CCXML session through a createsession event I/O processor:
	`http://localhost:8080/createsession?uri=`
	`  http://www.example.org/ccxml/myscript.ccxml&`
	`eventsource=http://www.example.org/ccxmlext&delay.value=500&`
	`delay.format=msecs&vxmlscript=`
	`  http://www.example.org/ccxml/myscript.vxml`
	The createsession event I/O processor uses the same web server as the basichttp event I/O. For more information, see Appendix L in http://www.w3.org/TR/2005/WD-ccxml-20050629.

Exchanging data between CCXML and VoiceXML

A CCXML application can send data to a VoiceXML application by using the ***namelist*** attribute of the <dialogprepare> or <dialogstart> element. A CCXML application can return data to a VoiceXML application by using the ***expr*** or ***namelist*** attributes (or both) in the BYE message generated as a result of the CCXML <exit> element.

A VoiceXML application can send data to a CCXML application in the BYE message that is generated as a result of the VoiceXML <disconnect> or <exit> element. The CCXML application can retrieve data in the CCXML dialog.exit or dialog.disconnect events.

CCXML event processing

The Vision Call Server processes CCXML events using the CCXML basichttp event processor. The basichttp event processor goes through the following stages:

Stage	Description
1	A web server listens on a particular port (8080 by default) for incoming HTTP requests.
2	Upon receiving a request, the web server extracts CCXML parameters and populates a related CCXML event.

3	If the event is destined for an active CCXML session, the web server injects the CCXML event into the CCXML session using a request like the following:
	`http://localhost:8080/basichttp?sessionid=s123&name=basichttp.myevent&`
	`eventsource=http://www.example.org/ccxmlext&agent=agent12&site.location=orlando&`
	`site.code=RE445`
	All requests created from CCXML events use /basichttp as the access URI.

Use the CCXML <send> element to send CCXML events to an external component.

For more information about the CCXML basichttp event processor, see Appendix K of the W3C Working Draft of *Voice Browser Call Control: CCXML Version 1.0*, dated 29 June 2005.

In addition, the Vision Call Server supports a createsession event I/O processor. An external component can start a new CCXML session as described in *Starting a new CCXML session* on page 13.

For information about the CCXML standard, see http://www.w3.org/TR/2005/WD-ccxml-20050629.

Additional attributes in the connection.alerting event

The CCXML connection.alerting event has the following proprietary attributes:

Attribute	Description
callmode	Indicates the type of incoming call. Valid values: • voice • video
callid	Specifies the SIP call ID of an incoming SIP call.

substate attribute in the connection.signal event

The substate attribute in the connection.signal event is set to ringing received when the called party's phone is ringing, and then set to dtmf-**x** when receiving a DTMF, where **x** is the DTMF value. This feature is available only if the dtmfEventNotification setting in the *telecom.conf* file is set to TRUE. (It is set to FALSE by default.)

The substate attribute is set to TRANSFER when receiving a transfer request from a SIP leg.

Reason mapping in the connection.failed event

When an inbound or outbound call fails to complete its connection, the connection.failed event is sent to the CCXML session with the reason attribute set to either:

- A SIP error code, if a SIP call fails, or
- The Q.850 cause value, if an ISDN or ISUP call fails.

CCXML logging

CCXML application logs are included in the Vision Call Server logs. By default, the CCXML application log level is 4. Change the log level by resetting the CcxmlAppLogLevel setting in the Vision Call Server configuration file (*callserver.conf*). For more information, see the *Dialogic® Vision™ Call Server Administration Manual*.

General CCXML scenarios

This topic describes how to implement the following scenarios using CCXML:

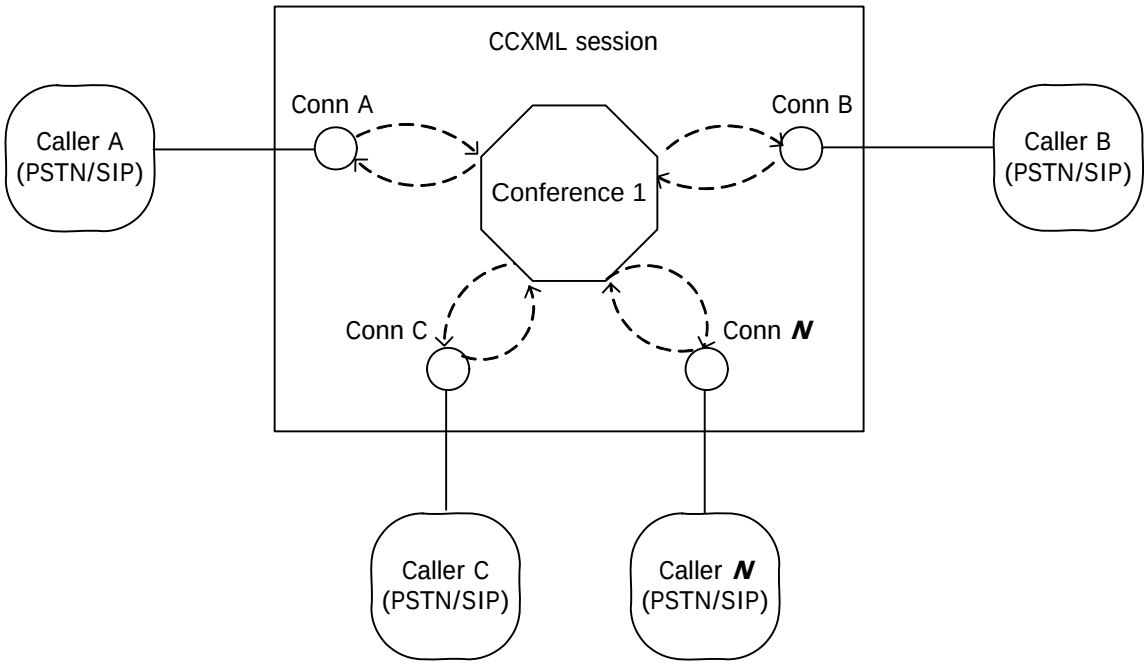
- Multiple party audio conferencing
- Media splitting for eavesdropping calls
- Third party call control
- Outbound SIP audio calls with preferred codecs

Multiple party audio conferencing

Multiple party audio conferencing describes the behavior by which a CCXML application creates a conference object for audio mixing. The conference object can have multiple audio inputs and one logical output. The output of this object is derived by mixing all of the input streams and then removing the audio contributed by the entity receiving the output. The maximum number of conference participants depends on the Vision™ hardware port density.

In a multiple party audio conference, callers, call center agents, and call center supervisors can connect through ISDN, ISUP, or SIP.

The following configuration shows multiple callers connected together in an ***N***-way conference call. Each caller hears the mixed output of the other callers:



To start the conference, Caller A creates the conference object, and limits the conference to 10 callers:

```
<createconference  conferenceid="confid"  confname="conferenceForN"  reservedtalkers="10"
reservedlisteners="10"
```

Each subsequent caller (up to nine) connects to the conference object using a join element with the confid attribute:

```
<join id1="confid"  id2="connid" duplex="full"/>
```

The sample CCXML applications that come with the Vision™ Server shows how to implement a variety of conference types, including a conference with call recording and a conference with music on hold. For information, see *Overview of CCXML sample applications* on page 23.

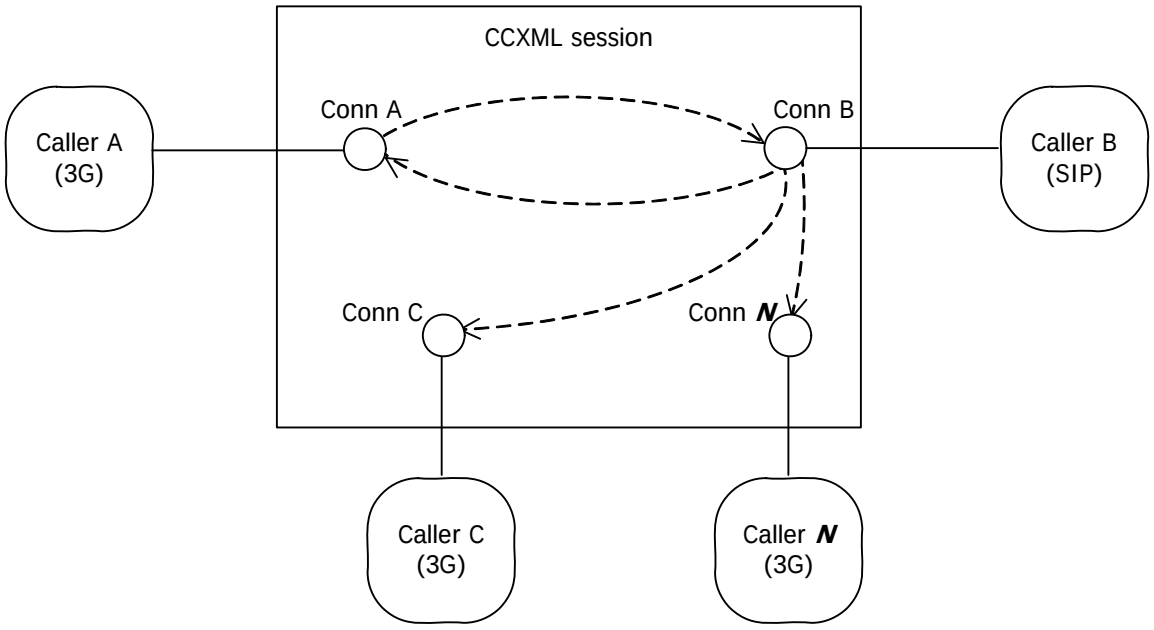
Media splitting for eavesdropping calls

Media splitting functionality allows one or more eavesdropping calls to listen in to a conversation between two other call endpoints, without being able to participate in that conversation. This type of scenario, for example, lets a supervisor listen in to a conversation between a user and an agent. In this scenario, the call endpoints engaged in a conversation are connected together in full duplex mode, while the listening call legs are connected in half duplex mode.

Use the parameter “mediasplit:'true'” in the hints attribute of the <join> element to implement media splitting. The “mediasplit:'true'” hint is mandatory in the join request for the two call endpoints engaged in a conversation. Media splitting is allowed only when one call endpoint has a 3G connection and the other has a SIP connection. The maximum number of connections in the same media split scenario is 32.

For example, suppose you have the following connections in a CCXML session:

- Caller A - 3G connection
- Caller B - SIP audio/video connection
- Caller C - 3G connection
- Caller ***N*** - 3G connection



To connect Caller A to Caller B so that output flows in both directions (A to B and B to A) use a join request that uses full duplex mode and requests media splitting:

```
<join id1="connectionidA" id2="connectionidB" duplex="'full'"
hints="mediasplit:'true'"/>
```

To connect Caller C to Caller B so that output flows from B to C, use a join request that uses half duplex mode:

```
<join id1="connectionidC" id2="connectionidB" duplex="'half'"/>
```

To connect Caller ***N*** to Caller B, so that output flows from B to ***N***, use another join request that uses half duplex mode:

```
<join id1="connectionid***N***" id2="connectionidB" duplex="'half'"/>
```

To see an example of CCXML scripts that implement an eavesdropping scenario, see *Third party call with transfer out* on page 36.

Third party call control

Third party call control occurs when the Vision Call Server sets up and manages a call between two other parties. Applications might require this functionality in order to provide operator and conferencing services.

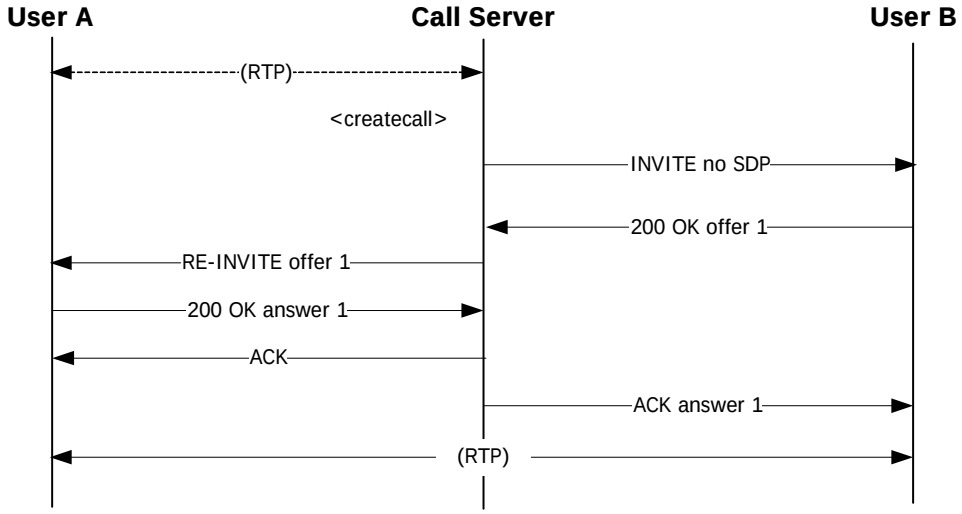
With third party call control, media flows between two SIP connections while the Vision Call Server remains in the signaling path. For example, suppose you have an established SIP connection called connection A, and you want to place an outbound SIP call using the SDP provided by connection A. To create the outbound call, use the following CCXML attributes and parameters:

- Join the two call endpoints using the <join> element.
- Specify third party control using the “tpcc:'true” parameter in the hints attribute of the <createcall> element:

```
<createcall dest="sip:987@10.1.2.16:5065" callerid="callingparty" timeout="20s"
connectionid="connB" joinid="connA" hints="{tpcc:'true'}">
```

The third party control scenario that the Vision™ Server uses follows Flow 1 in RFC 3725, "Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP)." The only difference is that in the Vision™ Server scenario, the SDP offer from User B is sent to User A in a RE-INVITE message. This gives User A time to reply to the Vision Call Server. User A responds by forwarding its SDP answer to User B in the ACK message.

The following illustration shows the call flow for providing third party call control:



Note: In this scenario, join operations are not allowed between User A and User B.

To see an example of CCXML scripts that implement a third party call with transfer out scenario, see *Third party call with transfer out* on page 36.

Outbound SIP calls with preferred audio codecs

When performing third party call control, an application might require that an outbound SIP call be placed with a specific audio codec. To create an outbound call with a preferred audio codec, use the following attributes and parameters in the <createcall> element:

- Join the two call endpoints using the joinid attribute.
- Specify the preferred codec using audiocodec parameter in the hints attribute of the <createcall> element.

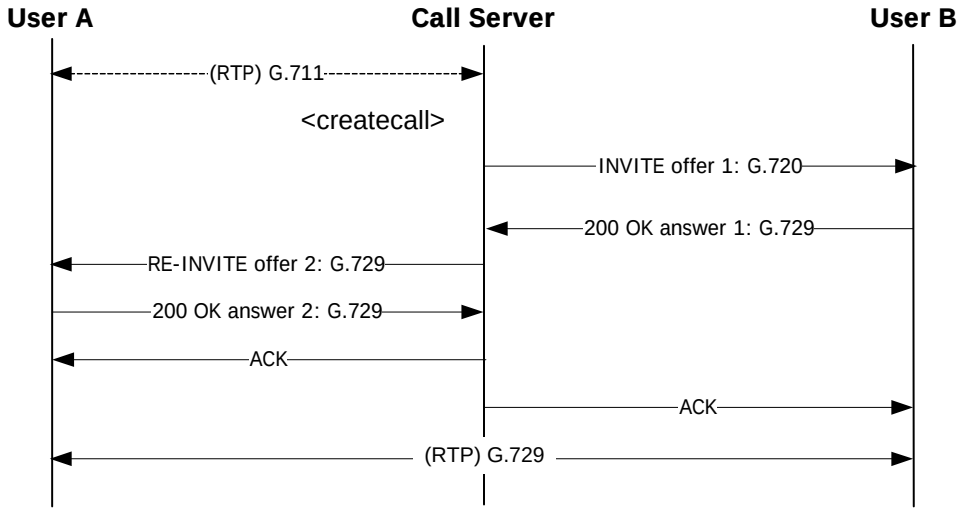
If the connection to join is a SIP connection that was previously established with another codec, the Vision Call Server sends a RE-INVITE message to negotiate the new codec. If the codecs negotiated in both connections are identical, the Vision Call Server does not send a RE-INVITE message.

Note: The audiocodec hint parameter cannot be used at the same time as the tpcc hint parameter.

The following example creates an outbound call with preferred codec G.729:

```
<createcall dest="'sip:987@10.1.2.16:5065'" callerid="callingparty" timeout="'20s'"
connectionid="connB" joinid="connA" hints="{audiocodec:'g729'}"/>
```

The following illustration shows the call flow for placing an outgoing SIP call with a preferred codec:



CCXML application definition file

The CCXML application definition file maps individual CCXML applications to number ranges that trigger the execution of those applications. If you need this mapping functionality, you can either use the Vision™ Console to set up an application definition, or you can edit the application definition file directly. By default, the CCXML application definition file is named *ccxmlappcfg.xml* and is located in the *vx/callserver/conf* directory.

This topic describes how to edit the application definition file directly. For information about using the Vision™ Console to change CCXML application definitions, see the *Dialogic® Vision™ CX Gateway Administration Manual* or the *Dialogic® Vision™ VX Integrated Media Platform User's Manual.*

The CCXML application definition file contains one <application> element for each CCXML application to be mapped. Each <application> element contains the following child elements:

Element	Description
<number-range>	Range of numbers that match to the application.
<script>	The type attribute contains the URI of the initial document that the Vision Call Server loads when the call is answered.

<dialog-route>	Comma-separated list of dialog servers to use for the CCXML application. Define each dialog server according to the following format:
	`<dialog-route audio-codec-pref="codec">`
	`IP_Address:Port[Priority]</dialog-route>`
	where:
	• ***codec*** is the preferred audio codec, if any. The audio-codec-pref attribute is optional.
	• ***IP_Address***:***Port*** is the IP address and port of the dialog server.
	• ***Priority*** is the priority level for load balancing over dialog servers. Priority is optional, and its value is relative from 0 (the default) to any required level. The highest priority is expressed by the lowest value (typically 0).
	Load balancing is performed between dialog servers that have the same priority and starts with dialog servers defined with the highest priority.
	Note: If the CCXML session was created with a CreateSesssion request, then no application definition is associated with the session. In this case, the CCXML engine uses the default dialog server specifed by the SIP_uas value in the *telecom.com* file to initiate the dialog. For information, see the *Dialogic® Vision™ Call Server Administration Manual*.

Element	Description
<outbound-route>	Comma-separated list of PSTN and SIP routes for outbound calls. Define each PSTN outbound route according to the following syntax: route-***Route_Number***[***Priority***] where: • ***Route_Number*** is the identifier of the PSTN route as defined in the *telecom.conf* file. • ***Priority*** is the priority level for load balancing over telecom routes. If ***Route_Number*** is set to 0, the outbound call is routed as specified by the associated incoming call. If the outbound call is not associated with an incoming call (such as when it is created with a CreateSession request), the default route is 1. Define each SIP outbound route according to the following syntax: ***IP_Address***:***Port***[***Priority***] where: • ***IP_Address:Port*** is the IP address and port of the SIP route. • ***Priority*** is the priority level for load balancing over SIP routes. Priority is optional and its value is relative from 0 (default) to any required level. The highest priority is expressed by the lowest value (typically 0). Load balancing is performed between routes (PSTN or SIP according to nature of the outbound call) with the same priority and starts by routes defined with the highest priority. The <outbound-route> element contains an optional audio-codec-pref attribute that specifies the audio codec to use when a SIP dialog is initiated. If a VoiceXML application associated with the CCXML application uses speech services, such as ASR and TTS, the corresponding <outbound-route> element must specify pcmu as the audio-codec-pref value.

The following CCXML application definition file maps application 1 to the file://c:/vx/callserver/www/ccxml/inbound.ccxml URI, and application 2 to the file://c:/vx/callserver/www/ccxml/outbound.ccxml URI:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<config>
<applications>

<application id="1">
<number-range>7768</number-range>
<script>file://c:/vx/callserver/www/ccxml/inbound.ccxml</script>
<dialog-route audio-codec-pref=”pcmu">10.1.2.16:5060[0]</dialog-route>
<outbound-route audio-codec-pref="pcmu">route-1[0]</outbound-route>
</application>

<application id="2">
<number-range>7769</number-range>
<script>file://c:/vx/callserver/www/ccxml/inbound2.ccxml</script>
<dialog-route audio-codec-pref="amr">10.0.0.106:5060[1],10.1.2.16:5060[0]</dialog-route>
<outbound-route audio-codec-pref="amr">route-1[0]</outbound-route>
</application>

<application id="3">
<number-range>7770</number-range>
<script>file://c:/vx/callserver/www/ccxml/outbound.ccxml</script>
<dialog-route audio-codec-pref="g729">10.0.0.3[0], 10.0.[3]</dialog-route>
<outbound-route>route-7[0], route-8[1]</outbound-route>
</application>

</applications>
</config>
```

4 CCXML sample applications

Overview of CCXML sample applications

All Vision™ Servers except IP-only audio models are installed with sample applications that use CCXML to create conferences and join users to them. The following table describes the sample applications:

Sample application	Description	For more information, see...
Call recording	Records the conversation between one user and one agent in a call center system.	*Call recording* on page 24
Music on hold	Places user calls on hold before they are transferred to agents, and streams music to calls while they are on hold.	*Music on hold* on page 27
Music on hold with a wait queue	Provides the same functionality as the music on hold application. In addition, this application places users in a wait queue when they call, and lets them know their position in the queue.	*Music on hold with a wait queue* on page 31
Third party call with transfer out	Provides both call recording functionality and music on hold functionality. In this application, an agent can place a user on hold while the agent contacts a supervisor. Afterwards, the application can reconnect the user to the agent, supervisor, or both.	*Third party call with transfer out* on page 36
Whisper coaching	Enables a supervisor to "whisper" advice to an agent without being heard by the user.	*Whisper coaching* on page 45

The sample applications consist of CCXML, VXML, and WAV files. The applications also require the use of web servers and application servers, which are not provided. A suitable option is Apache HTTP server 2.2 as the web server and CGI Perl script for the application server. The web server must be a destination that can accept and save multipart/form-data. The application server must be able to store a variable and return it to the CCXML script, when asked.

Configuring the sample applications

To configure the sample applications, follow these steps:

Step	Action
1	Copy the contents of *vx/demos/ccxml-demos/www* to *apache/htdocs*.
2	Copy the contents of *vx/demos/ccxml-demos/cgi-bin* to *apache/cgi-bin*.
3	For each use case, modify the VoiceXML and CCXML scripts to fit your network infrastructure, and define the application to the Vision Call Server. For information, see the topic for the use case.

Calculating resources

Vision™ Server models that include media boards provide DS0 and conferencing resources:

- DS0 resources convert endpoints from TDM streams to RTP streams, and RTP streams to TDM streams. One DS0 can convert one full-duplex endpoint. A SIP call uses one DSP resource. A PSTN calls does not use any DSP resources.
- Conferencing resources manage call legs in a conference object. One conferencing resource can manage one call leg in a conference object.

Depending on the number of media boards it contains, a typical Vision™ Server can support the following number of DS0 or conferencing resources:

Number of media boards	Number of DS0 resources supported	Number of CONF resources supported
1	240	448
2	480	896
3	720	1344

The conferencing implementation supports up to 128 members per conference object. Therefore, to achieve a desired member capacity, you might need to allocate multiple conference objects.

For information about the resources needed for implementing a specific use case, see the topic for that use case.

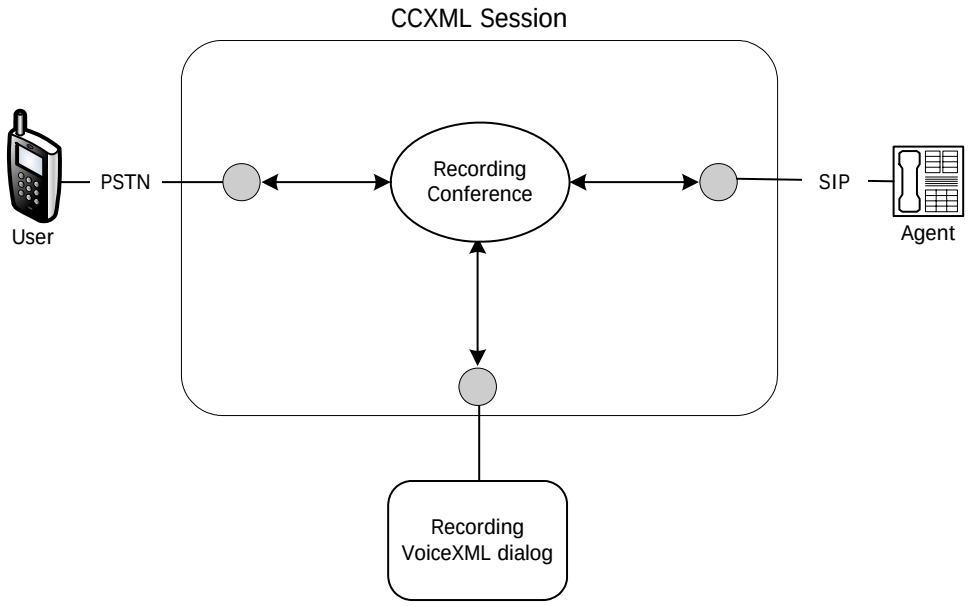
Call recording

The call recording sample application records the conversation between one user and one agent in a call center system.

This application uses the following files in the *vx/demos/ccxml-demos* directory:

File	Description
www/Call-Recording/incoming.ccxml	CCXML application that: • Joins a user and agent in a conference. • Attaches a VoiceXML dialog (*record.vxml*) to the conference for recording.
www/Call-Recording/record.vxml	VoiceXML application that records the conference.
cgi-bin/Call-Recording/SaveRecording.pl	CGI Perl script that accepts a POST from a VoiceXML record dialog.

The following illustration shows the connections created by the call recording application:



Resources and capacity

The call recording sample application uses three DS0 resources and three conferencing resources when the incoming user call is a SIP call, and two DS0 resources and three conferencing resources when the incoming user call is a PSTN call. The following table shows the tasks for which these resources are used:

Task	Resources needed
Initiating the VoiceXML recording dialog.	1 DS0
Accepting an incoming user call.	1 DS0 (for a SIP call only)
Contacting the agent.	1 DS0
Creating a three-party conference that consists of the user, agent, and call recording dialog.	3 conferencing

The following table shows the available port capacity for the call recording use case with a typical one-board, two-board, or three-board Vision™ Server configuration:

Number of boards	Network protocol for user calls	Port capacity
1	PSTN	120
	SIP	80
2	PSTN	240
	SIP	160
3	PSTN	360
	SIP	240

Running the application

The following table describes how to run the call recording application. These instructions assume that you already copied the contents of *vx/demos/ccxml-demos/www* to *apache/htdocs*, and *vx/demos/ccxml-demos/cgi-bin to apache/cgi-bin*, as described in *Overview of CCXML sample applications* on page 23.

Step	Action

1	Modify the *vx/demos/ccxml-demos/www/Call-Recording/incoming.ccxml* file as follows:
	• Change ***webserver*** to the URL of the web server that serves the *vx/demos/ccxml-demos/www/Call-Recording/record.vxml* file:
	`<dialogstart dialogid="dialogid" src="'http://webserver/Call-Recording/record.vxml'"`
	• Change ***sip:agent*** to specify the SIP URI for the agent:
	`<createcall dest="'sip:agent'"`

2	In the *vx/demos/ccxml-demos/www/Call-Recording/record.vxml* file, change the value of ***webserver*** to specify the web server that serves the *vx/demos/ccxml-demos/www/Call-Recording/saveRecroding.pl* file: `src="http://webserver/cgi-bin/Call-Recording/saveRecording.pl"`
3	In the Vision™ Console, define the call recording application to the Vision Call Server by associating a phone number with the following URL: `http://webserver/Call-Recording/incoming.ccxml` where ***webserver*** is the URL of the web server that serves the *vx/demos/ccxml-demos/www/Call-Recording/incoming.ccxml* file. For information about defining CCXML applications using the Vision™ Console, see the *Dialogic® Vision™ CX Video Gateway Administration Manual* or the *Dialogic® Vision™ VX Integrated Media Platform User's Manual.* For information about defining CCXML applications directly in the application definition file, see *CCXML application definition file* on page 19.
4	Make a call to the number you specified in Step 3.
5	After the scripts run, ensure that the audio was played back to the audio terminal.

Application logic

The call recording sample application follows this logic, which occurs in the *incoming.ccxml* file:

Step	Action	Code snippet
1	Create a three-party call recording conference.	`<createconference confname="session.id" conferenceid="conferenceid" reservedtalkers="3" reservedlisteners="0"/>`
2	Create a dialog that uses *recording.vxml* to record the conference.	`<dialogstart dialogid="dialogid" src="'http://webserver/Call-Recording/record.vxml'" conferenceid="conferenceid"/>`
3	Place a SIP call to an agent, and connect the agent to the call recording conference.	`<createcall dest="'sip:agent'" connectionid="AgentConnectionid" joinid="conferenceid"/>`
4	Accept an incoming user call.	`<accept connectionid="UserConnectionid"/>`
5	Connect the user call to the call recording conference. At this point, the user and agent can communicate, and the call is being recorded.	`<join id1="conferenceid" id2="UserConnectionid" entertone="false" exittone="false"/>`
6	Upon receipt of the connection.disconnected event, end the call recording conference by closing the CCXML session.	`<transition state="Active" event="connection.disconnected" name="evt"> <log label="'+++Call Recording Use Case+++'" expr="'Connection Disconnected. Exiting.'"/> <exit/> </transition>`

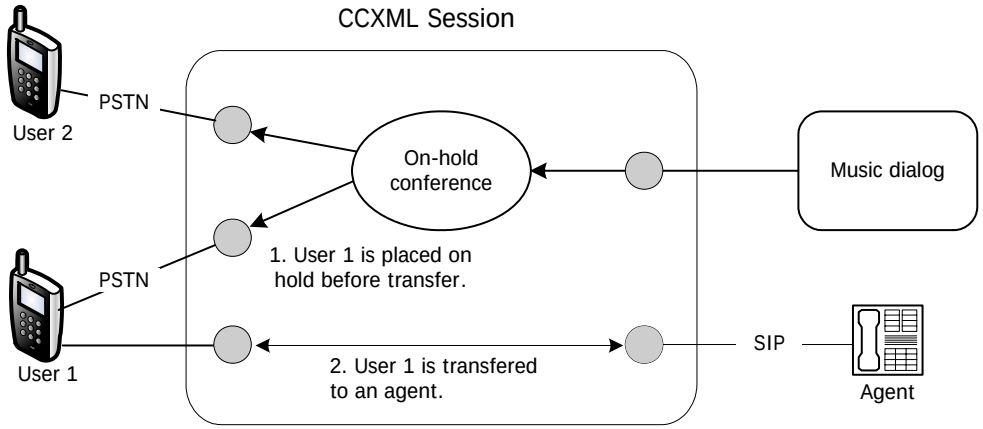
Music on hold

The music on hold sample application puts user calls on hold before they are transferred to an agent. The application streams music to calls while they are on hold.

This application uses the following files in the *vx/demos/ccxml-demos/www/Music-On-Hold* directory:

File	Description
create-MusicOnHold.ccxml	CCXML application that: • Creates a multi-user conference for incoming calls. • Invokes a VoiceXML dialog to play music while the users are on hold. • Releases the call from the conference, once the application transfers the call to an agent.
incoming.ccxml	CCXML application that accepts incoming calls and joins them to the conference.
music.vxml	VoiceXML application that streams music to the calls in the conference, while they wait for a transfer to an agent.
music.wav	WAV file containing the music that is played.

The following illustration shows the connections that the music on hold application creates:



Resources and capacity

The music on hold sample application uses three DS0 resources and 120 conferencing resources when the incoming user call is a SIP call, and two DS0 resources and 120 conferencing resources when the incoming user call is a PSTN call The following table shows the tasks for which these resources are used:

Task	Resources needed
Initiating the VoiceXML recording dialog.	1 DS0
Accepting an incoming user call.	1 DS0 (for a SIP call only)
Contacting the agent.	1 DS0
Creating a conference that can hold 1 talker and up to 119 listeners. The talker is the music dialog. The listeners include users and agents.	120 conferencing

The following table shows the available port capacity for the call recording use case with a typical one-board, two-board, or three-board Vision™ Server configuration:

Number of boards	Network protocol for user calls	Port capacity
1	PSTN	120
	SIP	119
2	PSTN	240
	SIP	238
3	PSTN	360
	SIP	357

Running the application

The following table describes how to run the music on hold application. These instructions assume that you already copied the contents of *vx/demos/ccxml-demos/www* to *apache/htdocs*, and *vx/demos/ccxml-demos/cgi-bin* to *apache/cgi-bin*, as described in *Overview of CCXML sample applications* on page 23.

Step	Action

1	Modify the *vx/demos/ccxml-demos/www/Music-On-Hold/create-MusicOnHold.ccxml* file as follows:
	• Change ***webserver*** to the URL of the web server that serves the *vx/demos/ccxml-demos/www/Music-On-Hold/music.vxml* file:
	`<dialogstart src="'http://webserver/Music-On-Hold/music.vxml'"`
	• Change ***appserver*** to the application server on which to post the conference ID for the music on hold conference:
	`<send target="'http://appserver/MusicOnHold'"`

2	Modify the *vx/demos/ccxml-demos/www/Music-On-Hold/incoming.ccxml* file as follows:
	• Change ***appserver*** to the application server from which to obtain the conference ID for the music on hold conference:
	`<send target="'http://appserver/MusicOnHold'"`
	• Change ***sip:agent*** to specify the SIP URI for the agent:
	`<createcall dest="'sip:agent'"`

3	Start the music on hold conference by entering the following URL in your web browser: `http://Server-IP:8080/createsession?uri=http://webserver /Music-On-Hold/create-MusicOnHold.ccxml` where ***Server-IP*** is the IP address of the Vision™ Server. This URL executes the *create-MusicOnHold.ccxml* file, which creates the conference.
4	In the Vision™ Console, define the music on hold application to the Vision Call Server by associating a phone number with the following URL: `http://webserver/Music-On-Hold/incoming.ccxml` where ***webserver*** is the URL of the web server that serves the *vx/demos/ccxml-demos/www/Music-On-Hold/incoming.ccxml* file. For information about defining CCXML applications using the Vision™ Console, see the *Dialogic® Vision™ CX Video Gateway Administration Manual* or the *Dialogic® Vision™ VX Integrated Media Platform User's Manual.* For information about defining CCXML applications

Step	Action
	directly in the application definition file, see *CCXML application definition file* on page 19.
5	Make a call to the number you specified in Step 4.
6	After the scripts run, ensure that the audio was played back to the audio terminal.

Note: The *music.vxml* file plays once and then returns. To play music for a longer period of time, loop the *music.vxml* file when it finishes.

Application logic

The music on hold sample application follows this logic:

Step	Action	File name and code snippet
1	Create a music on hold conference that supports one talker and up to 120 listeners.	*create-MusicOnHold.ccxml* `<createconference confname="'onHoldConference'" conferenceid="conferenceid" reservedtalkers="1" reservedlisteners="120"/>`
2	Create a dialog that uses *music.vxml* to play music to music on hold conference members.	*create-MusicOnHold.ccxml* `<dialogstart src="'http://webserver/Music-On-Hold/music.vxml'" conferenceid="conferenceid"/>`
3	Publish the music on hold conference ID to an application server.	*create-MusicOnHold.ccxml* `<send target="'http://appserver/MusicOnHold'" targettype="'basichttp'" data="'setConferenceInfo'" namelist="basichttpioprocessor conferenceid"/>`

4	Accept an incoming user call.	*incoming.ccxml*
		`<assign name="UserConnectionid" expr="evt.connectionid"/>`
		`<accept connectionid="UserConnectionid"/>`

5	Fetch the conference ID.	*incoming.ccxml* `<send target="'http://appserver/MusicOnHold'" targettype="'basichttp'" data="'getConferenceInfo'" namelist="basichttpioprocessor session.id"/>`
6	Connect an incoming call to the music on hold conference using a half-duplex connection.	*incoming.ccxml* `<join id1="UserConnectionid" id2="onHoldConferenceid" duplex="'half'" entertone="false" exittone="false"/>`

7	Upon receipt of the application.RemoveFromHold event, remove the user from the music on hold conference.	*incoming.ccxml*
		`<transition state="onHold" event="application.RemoveFromHold" name="evt">`
		`<log label="'+++Music-On-Hold Use Case+++'" expr="'Removing User from hold'"/>`
		`<unjoin id1="UserConnectionid" id2="onHoldConferenceid"/>`
		`<assign name="state" expr="'ConnectingAgent'"/>`
		`</transition>`

8	Place a SIP call to an agent, and join the call to the user.	*incoming.ccxml* `<createcall dest="'sip:agent'" connectionid="AgentConnectionid" joinid="UserConnectionid"/>`
9	Upon receipt of the connection.disconnected	*incoming.ccxml* `<transition event="connection.disconnected"`

Step	Action	File name and code snippet

	event, end the music on hold conference by closing the CCXML session.	`name="evt">`
		`<log label="'+++Music-On-Hold Use Case+++'"`
		`expr="'Connection disconnected. Exiting.'"/>`
		`<exit/>`
		`</transition>`

Music on hold with a wait queue

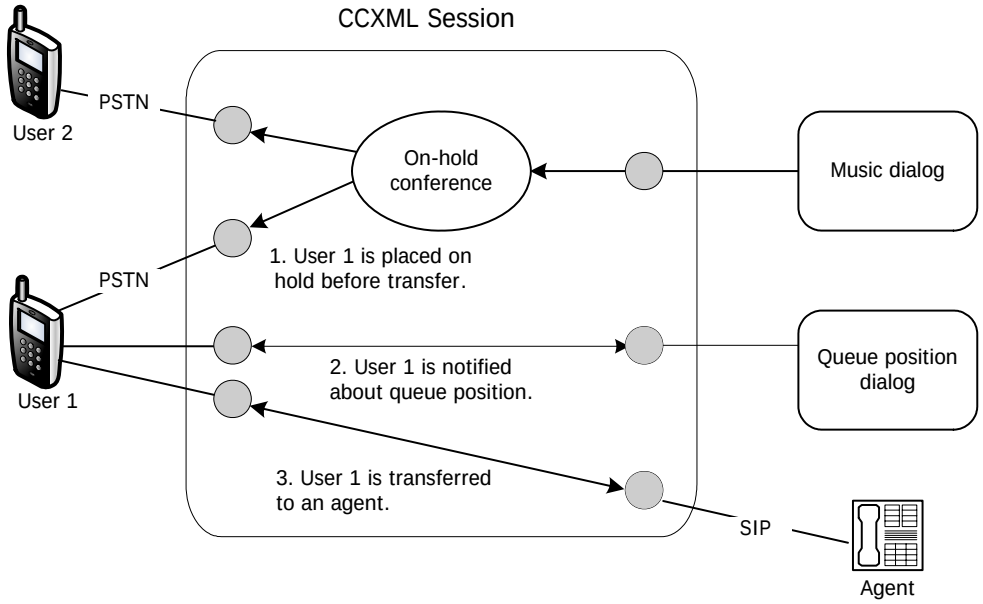
The music on hold with a wait queue sample application has the same functionality as the music on hold application, except that it lets users know their position in the on-hold queue.

Note: For demonstration purposes, the queue value is hard coded.

This application uses the following files in the *vx/demos/ccxml-demos/www/Music-On-Hold* directory:

File	Description
create-MusicOnHold.ccxml	CCXML application that: • Creates a multi-user music on hold conference for incoming calls. • Invokes a VoiceXML dialog to play music while the users are on hold.
incoming-queue.ccxml	CCXML application that: • Accepts incoming calls. • Joins incoming calls to the music on hold conference. • Periodically creates a dialog that notifies users of their position in the queue. • Releases the call from the music on hold conference, and transfers the call to an agent.
music.vxml	VoiceXML application that streams music to the calls in the music on hold conference, while they wait for a transfer to an agent.
queuePosition.vxml	VoiceXML application that plays the current position of the user in the queue. (The user position is hard coded with the value "2nd" in this sample file.)
music.wav	WAV file containing the music played by *music.vxml*.
2.wav *intro.wav* *outro.wav*	WAV files containing the queue information played by *queuePosition.vxml*.

The following illustration shows the connections that the music on hold with a wait queue application creates:



Resources and capacity

The music on hold with a wait queue sample application uses three DS0 resources and 120 conferencing resources when the incoming user call is a SIP call, and two DS0 resources and 120 conferencing resources when the incoming user call is a PSTN call The following table shows the tasks for which these resources are used:

Task	Resources needed
Initiating the VoiceXML recording dialog.	1 DS0
Accepting an incoming user call.	1 DS0 (for a SIP call only)
Contacting the agent.	1 DS0
Creating a conference that can hold 1 talker and up to 119 listeners. The talker is the music dialog. The listeners include users and agents.	120 conferencing

The following table shows the available port capacity for the call recording use case with a typical one-board, two-board, or three-board Vision™ Server configuration:

Number of boards	Network protocol for user calls	Port capacity
1	PSTN	120
	SIP	119
2	PSTN	240
	SIP	238
3	PSTN	360
	SIP	357

Running the application

The following table describes how to run the music on hold with a wait queue application. These instructions assume that you already copied the contents of *vx/demos/ccxml-demos/www* to *apache/htdocs*, and *vx/demos/ccxml-demos/cgi-bin to apache/cgi-bin*, as described in *Overview of CCXML sample applications* on page 23:

Step	Action

1	Modify the *vx/demos/ccxml-demos/www/Music-On-Hold/create-MusicOnHold.ccxml* file as follows:
	• Change ***webserver*** to the URL of the web server that serves the *vx/demos/ccxml-demos/www/Music-On-Hold/music.vxml* file:
	`<dialogstart src="'http://webserver/Music-On-Hold/music.vxml'"`
	• Change ***appserver*** to the application server on which to post the conference ID for the music on hold conference:
	`<send target="'http://appserver/MusicOnHold'"`

2	Modify the *vx/demos/ccxml-demos/www/Music-On-Hold/incoming-queue.ccxml* file as follows:
	• Change ***appserver*** to the application server from which to obtain the conference ID for the music on hold conference:
	`<send target="'http://appserver/MusicOnHold'"`
	• Change ***webserver*** to the URL of the web server that serves the *vx/demos/ccxml-demos/www/Music-On-Hold/queuePosition.vxml* file:
	`<dialogstart src="'http://webserver/Music-On-Hold/vxml/queuePosition.vxml'"`
	• Change ***sip:agent*** to specify the SIP URI for the agent:
	`<createcall dest="'sip:agent'"`

3	Start the music on hold conference by entering the following URL in your web browser: `http://Server-IP:8080/createsession?uri=http://webserver/Music-On-Hold/create-MusicOnHold.ccxml` where ***Server-IP*** is the IP address of the Vision™ Server. This URL executes the *create-MusicOnHold.ccxml* file, which creates the conference.
4	In the Vision™ Console, define the music on hold with a wait queue application to the Vision Call Server by associating a phone number with the following URL:

Step	Action
	`http://webserver/Music-On-Hold/incoming-queue.ccxml` where ***webserver*** is the URL of the web server that serves the *vx/demos/ccxml-demos/www/Music-On-Hold/incoming-queue.ccxml* file. For information about defining CCXML applications using the Vision™ Console, see the *Dialogic® Vision™ CX Video Gateway Administration Manual* or the *Dialogic® Vision™ VX Integrated Media Platform User's Manual.* For information about defining CCXML applications directly in the application definition file, see *CCXML application definition file* on page 19.
5	Make a call to the number you specified in Step 3.
6	After the scripts run, ensure that the audio was played back to the audio terminal.

Note: The *music.vxml* file plays one time and then returns. To play music for a longer period of time, loop the *music.vxml* file when it finishes.

Application logic

The music on hold with a wait queue application follows this logic:

Step	Action	File name and code snippet
1	Create a music on hold conference that supports one talker and up to 120 listeners.	*create-MusicOnHold.ccxml* `<createconference confname="'onHoldConference'" conferenceid="conferenceid" reservedtalkers="1" reservedlisteners="120"/>`
2	Create a dialog that uses *music.vxml* to play music to music on hold conference members.	*create-MusicOnHold.ccxml* `<dialogstart src="'http://webserver/Music-On-Hold/music.vxml'" conferenceid="conferenceid"/>`
3	Publish the music on hold conference ID to an application server.	*create-MusicOnHold.ccxml* `<send target="'http://appserver/MusicOnHold'" targettype="'basichttp'" data="'setConferenceInfo'" namelist="basichttpioprocessor conferenceid"/>`
4	Create the music on hold conference	Create the music on hold conference by entering the following URL in your web browser: `http://CallServer-IP:8080/createsession?uri=http://webserver/Music-On-Hold/create-MusicOnHold.ccxml` where ***Callserver_IP*** is the IP address of the Vision Call Server. This URL tells the CCXML engine to execute the *create-MusicOnHold.ccxml* file, which creates the music on hold conference.

5	Accept an incoming user call, and send the user's queue position to the application server after 60 seconds.	*incoming-queue.ccxml*
		`<assign name="UserConnectionid" expr="evt.connectionid"/>`
		`<accept connectionid="UserConnectionid"/>`
		`<send target="session.id" targettype="'ccxml'" data="'application.inputQueuePosition'" delay="'60s'"/>`

6	Fetch the music on hold conference ID.	*incoming-queue.ccxml* `<send target="'http://appserver/MusicOnHold'"`

Step	Action	File name and code snippet
		`targettype="'basichttp'" data="'getConferenceInfo'" namelist="basichttpioprocessor session.id"/>`
7	Connect the incoming call to the music on hold conference using a half-duplex connection.	*incoming-queue.ccxml* `<join id1="UserConnectionid" id2="onHoldConferenceid" duplex="'half'" entertone="false" exittone="false"/>`
8	Upon receipt of the application.inputQueuePosition event, remove the user from the music on hold conference.	*incoming-queue.ccxml* `<transition state="onHold" event="application.inputQueuePosition" name="evt"> <log label="'+++Music-On-Hold Use Case+++'" expr="'We need to tell the user position in the queue.'"/> <unjoin id1="UserConnectionid" id2="onHoldConferenceid"/> <assign name="state" expr="'inputQueuePosition'"/> </transition>`
9	Create a dialog that joins *queuePostion.vxml* to the call. The dialog plays the user's position in the queue and then exits.	*incoming-queue.ccxml* `<dialogstart src="'http://webserver/Music-On-Hold/vxml/queuePosition.vxml'" connectionid="UserConnectionid"/>`
10	Reconnect the user to the music on hold conference.	*incoming-queue.ccxml* `<join id1="UserConnectionid" id2="onHoldConferenceid" duplex="'half'" entertone="false" exittone="false"/>`
11	Send the user's queue position to the application server every 60 seconds.	*incoming-queue.ccxml* `<send target="session.id" targettype="'ccxml'" data="'application.inputQueuePosition'" delay="'60s'"/>`
12	Upon receipt of the application.RemoveFromHold event, remove the user from the music on hold conference.	*incoming-queue.ccxml* `<transition state="onHold" event="application.RemoveFromHold" name="evt"> <log label="'+++Music-On-Hold Use Case+++'" expr="'Removing User from hold'"/> <unjoin id1="UserConnectionid" id2="onHoldConferenceid"/> <assign name="state" expr="'ConnectingAgent'"/> </transition>`
13	Place a SIP call to an agent, and join the call to the user.	*incoming-queue.ccxml* `<createcall dest="'sip:agent'" connectionid="AgentConnectionid" joinid="UserConnectionid"/>`
14	Upon receipt of the connection.disconnected event, end the music on hold conference by closing the CCXML session.	*incoming.ccxml* `<transition event="connection.disconnected" name="evt"> <log label="'+++Music-On-Hold Use Case+++'" expr="'Connection disconnected. Exiting.'"/> <exit/> </transition>`

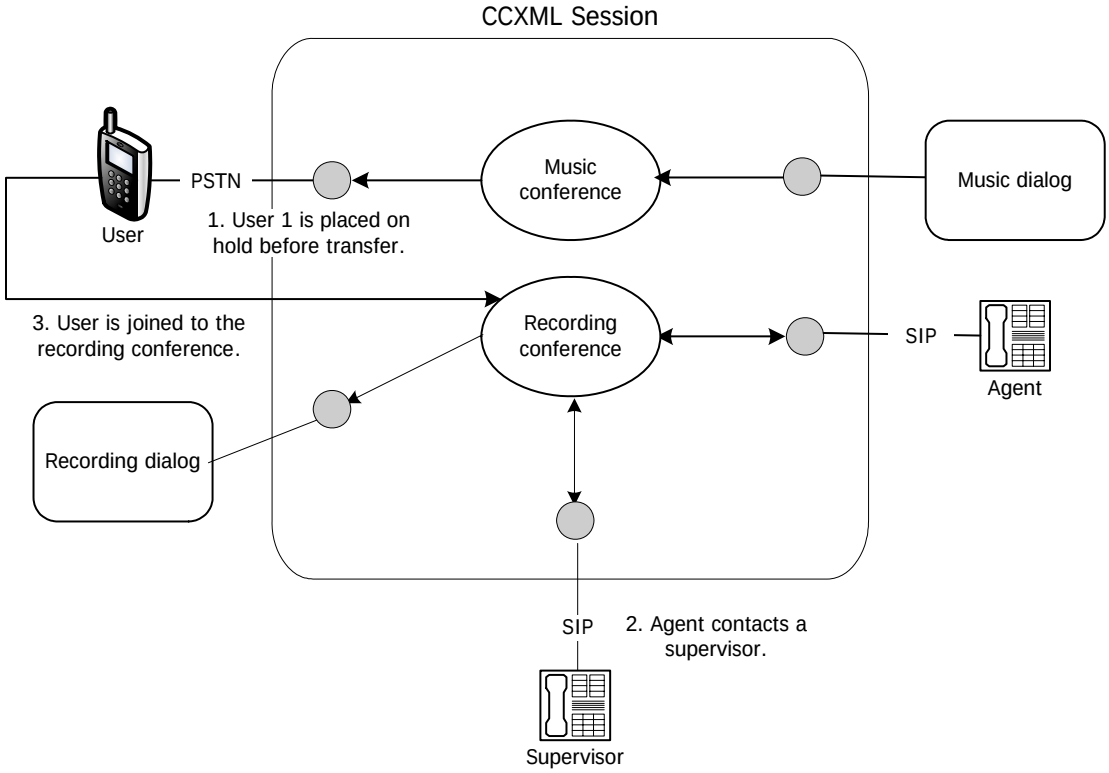
Third party call with transfer out

The third party call with transfer out application provides both call-recording functionality and music on hold functionality. In this application, an agent can place a user call on hold while the agent contacts a supervisor. After the agent and supervisor are connected, the application can join the user to the agent, the user to the supervisor, or the user to both the agent and supervisor.

This application uses the following files in the *vx/demos/ccxml-demos* directory:

File	Description
www/Music-On-Hold/create-MusicOnHold.ccxml	CCXML application that: • Creates a multi-user music on hold conference for incoming calls. • Invokes a VoiceXML dialog to play music while the users are on hold.
www/Third-Party-Call/incoming.ccxml	CCXML application that: • Creates a call recording conference. • Accepts incoming calls. • Joins incoming calls to the music on hold conference. • Releases the call from the music on hold conference, and transfers the call to the call recording conference, where the caller can speak to an agent, supervisor, or both.
www/Music-On-Hold/music.vxml	VoiceXML application that streams music to the calls in the music on hold conference, while they wait for a transfer to an agent.
www/Music-On-Hold/music.wav	WAV file containing the music that is streamed.
cgi-bin/Call-Recording/SaveRecording.pl	CGI Perl script that accepts a POST from a VoiceXML record dialog.

The following illustration shows the connections that the third party call with transfer out application creates:



Resources and capacity

The third party call with transfer out sample application uses four DS0 resources and three conferencing resources when the incoming user call is a SIP call, and one DS0 resource and three conferencing resources when the incoming user call is a PSTN call. The following table shows the tasks for which these resources are used:

Task	Resources needed
Initiating the VoiceXML recording dialog.	1 DS0
Accepting an incoming user call.	1 DS0 (for a SIP call only)
Contacting the agent.	1 DS0
Contacting the supervisor.	1 DS0
Create a three-party conference that consists of the user, agent, and call recording dialog.	3 conferencing

Note: The SIP call leg that initiates the VoiceXML music dialog also uses one DS0 resource. However, because this occurs only once, when the first incoming call is attached to the music conference, it is not included in the total calculations.

The following table shows the available port capacity for the call recording use case with a typical one-board, two-board, or three-board Vision™ Server configuration. In the best case, no user is connected to a supervisor. In the worst case, all users are connected to supervisors.

Number of boards	Network protocol for user calls	Port capacity	
		Best case	Worst case
1	PSTN	82	79
	SIP	79	59
2	PSTN	164	258
	SIP	158	118
3	PSTN	246	237
	SIP	237	177

Running the application

The following table describes how to run the third party call with transfer out application. These instructions assume that you already copied the contents of *vx/demos/ccxml-demos/www* to *apache/htdocs*, and *vx/demos/ccxml-demos/cgi-bin to apache/cgi-bin*, as described in *Overview of CCXML sample applications* on page 23.

Step	Action
1	In the *vx/demos/ccxml-demos/www/Call-Recording/record.vxml* file, change the value of ***webserver*** to specify the web server that serves the *x/demos/ccxml-demos/www/Call-Recording/saveRecroding.pl* file: `src="http://webserver/cgi-bin/Call-Recording/saveRecording.pl"`

2	Modify the *vx/demos/ccxml-demos/www/Music-On-Hold/create-MusicOnHold.ccxml* file as follows:
	• Change ***webserver*** to the URL of the web server that serves the *vx/demos/ccxml-demos/www/Music-On-Hold/music.vxml* file:
	`<dialogstart src="'http://webserver/Music-On-Hold/music.vxml'"`
	• Change ***appserver*** to the application server on which to post the conference ID for the music on hold conference:
	`<send target="'http://appserver/MusicOnHold'"`

4	Modify the vx*/demos/ccxml-demos/Third-Party-Call/incoming.ccxml* file as follows:
	• Change ***webserver*** to the URL of the web server that serves the *vx/demos/ccxml-demos/www/Call-Recording/record.vxml* file:
	`<dialogstart dialogid="dialogid" src="'http://webserver/cnf_usecases/Call-Recording/record.vxml'"`
	• Change ***sip:agent*** to specify the SIP URI for the agent:
	`<createcall dest="'sip:agent'"`
	• Change ***appserver*** to the application server from which to obtain the conference ID for the music on hold conference:
	`<send target="'http://appserver/MusicOnHold'"`

Step	Action
5	Start the music on hold conference by entering the following URL in your web browser: `http://`***`Server-IP`***`:8080/createsession?uri=http://webserver /Music-On-Hold/create-MusicOnHold.ccxml` where ***Server-IP*** is the IP address of the Vision™ Server. This URL executes *create-MusicOnHold.ccxml*, which creates the conference.
6	In the Vision™ Console, define the third party call with transfer out call handler application to the Vision Call Server by associating a phone number with the following URL: `http://`**`webserver`**`/Third-Party-Call/incoming.ccxml` where ***webserver*** is the URL of the web server that serves the *vx/demos/ccxml-demos/www/Third-Party-Call/incoming.ccxml* file. For information about defining CCXML applications using the Vision™ Console, see the *Dialogic® Vision™ CX Video Gateway Administration Manual* or the *Dialogic® Vision™ VX Integrated Media Platform User's Manual*. For information about defining CCXML applications directly in the application definition file, see *CCXML application definition file* on page 19.
7	Make a call to the number you specified in Step 6.
8	After the scripts run, ensure that the audio was played back to the audio terminal.

Note: The *music.vxml* file plays once and then returns. To play music for a longer period of time, loop the *music.vxml* file when it finishes.

Application logic

The third party call with transfer out application follows this logic:

Task	Description	For more information see...
1	Create the music on hold conference.	*Task 1: Creating the music on hold conference* on page 40.
2	Create the call recording conference.	*Task 2: Creating the call recording conference* on page 40.
3	Connect an agent and user to the call recording conference.	*Task 3: Connecting an agent and user to the call recording conference* on page 40.
4	Place the user on hold by connecting the user to the music on hold conference.	*Task 4: Placing the user on hold* on page 41.
5	Connect a supervisor to the agent in the call recording conference.	*Task 5: Connecting a supervisor to the agent* on page 41.
6	Depending on the received event, the application can: • Re-join the user to the call recording conference, so the user can speak to the agent and supervisor simultaneously. • Remove the supervisor from the conference, so the user can speak only to the agent • Remove the agent from the conference, so the user can speak only to the supervisor.	*Task 6: Responding to different events* on page 41.
7	End the call recording conference.	*Task 7: Responding to different disconnect scenarios* on page 42

Task 1: Creating the music on hold conference

The application creates the music on hold conference:

Step	Action	File name and code snippet
1	Create an music on hold conference that supports one talker and up to 80 listeners.	*create-MusicOnHold.ccxml* `<createconference confname="'onHoldConference'" conferenceid="conferenceid" reservedtalkers="1" reservedlisteners="80"/>`
2	Create a dialog that uses *music.vxml* to play music to conference members.	*create-MusicOnHold.ccxml* `<dialogstart src="'http://webserver/Music-On-Hold/music.vxml'" conferenceid="conferenceid"/>`
3	Publish the conference ID to an application server.	*create-MusicOnHold.ccxml* `<send target="'http://appserver/MusicOnHold'" targettype="'basichttp'" data="'setConferenceInfo'" namelist="basichttpioprocessor conferenceid"/>`

Task 2: Creating the call recording conference

The application creates the call recording conference:

Step	Action	File name and code snippet
1	Create a call recording conference that supports up to four talkers and no listeners.	*incoming.ccxml* `<createconference confname="session.id" conferenceid="conferenceid" reservedtalkers="4" reservedlisteners="0"/>`

Task 3: Connecting an agent and user to the call recording conference

The application places a SIP call to connect an agent to the call recording conference. It also connects an incoming user call to this conference:

Step	Action	File name and code snippet
1	Place a SIP call to an agent, and connect the agent to the call recording conference.	*incoming.ccxml* `<createcall dest="'sip:agent'" connectionid="AgentConnectionid" joinid="conferenceid"/>`
2	Accept an incoming user call.	*incoming.ccxml* `<accept connectionid="UserConnectionid"/>`
3	Connect the user call to the call recording conference.	*incoming.ccxml* `<join id1="conferenceid" id2="UserConnectionid" entertone="false" exittone="false"/>`

At this point, the user is connected to the agent, and the call is recorded.

Task 4: Placing the user on hold

The application removes the user from the call recording conference, and joins the user to the music on hold conference:

Step	Action	File name and code snippet
1	Remove the user from the call recording conference.	*incoming.ccxml* `<unjoin id1="UserConnectionid" id2="conferenceid"/>`
2	Fetch the ID of the on hold conference.	*incoming.ccxml* `<send target="'http://appserver/MusicOnHold'" targettype="'basichttp'" data="'getConferenceInfo'" namelist="basichttpioprocessor session.id"/>`
3	Connect the user call to the music on hold conference, using a half-duplex connection.	*incoming.ccxml* `<join id1="UserConnectionid" id2="onHoldConferenceid" duplex="'half'" entertone="false" exittone="false"/>`

Task 5: Connecting a supervisor to the agent

While the user is on hold, the application connects the supervisor to an agent:

Action	File name and code snippet
Place a SIP call to a supervisor, and connect the supervisor to the call recording conference.	*incoming.ccxml* `<createcall connectionid="SupervisorConnectionid" dest="'sip:supervisor'" joinid="conferenceid"/>`

Task 6: Responding to different events

Depending on the event received, the application can take one of the following actions:

- Join the user to the call recording conference, so the user, agent, and supervisor can speak together.
- Remove the supervisor from the call recording conference, so the user can speak only to the agent.
- Remove the agent from the call recording conference, so the user can speak only to the supervisor.

Joining the user to the call recording conference:

The application follows these steps to join the user to the call recording conference:

Step	Action	Code snippet
1	Upon receiving the application.joinUser event, remove the user from the music on hold conference.	`<transition state="RecordingSupervisorAgent" event="application.joinUser" name="evt"> <log label="'+++Third Party Call Use Case+++'" expr="'Received request to join the user back to the supervisor and the agent.'"/> <assign name="state" expr="'JoiningUserToAgentSupervisor'"/> <unjoin id1="onHoldConferenceid" id2="UserConnectionid"/> </transition>`
2	Join the user to the call recording conference.	`<join id1="conferenceid" id2="UserConnectionid" entertone="false" exittone="false"/>`

Joining the user back to the agent

The application follows these steps to remove the supervisor from the call recording conference and join the user to the conference:

Step	Action	Code snippet
1	Upon receiving the application.unjoinSupervisor event, remove the supervisor from the call recording conference.	`<transition state="RecordingSupervisorAgent" event="application.unjoinSupervisor" name="evt"> <log label="'+++Third Party Call Use Case+++'" expr="'Received request to join the user back to the agent. Unjoining Supervisor'"/> <assign name="state" expr="'UnjoiningSupervisor'"/> <unjoin id1="conferenceid" id2="SupervisorConnectionid"/> </transition>`
2	Remove the user from the music on hold conference.	`<unjoin id1="onHoldConferenceid" id2="UserConnectionid"/>`
3	Join the user to the call recording conference.	`<join id1="conferenceid" id2="UserConnectionid" entertone="false" exittone="false"/>`

Joining the user to the supervisor

The application follows these steps to remove the agent from the call recording conference and join the user to the conference:

Step	Action	Code snippet
1	Upon receiving the application.unjoinAgent event, remove the agent from the call recording conference.	`<transition state="RecordingSupervisorAgent" event="application.unjoinAgent" name="evt"> <log label="'+++Third Party Call Use Case+++'" expr="'Received request to join the user to the supervisor agent. Unjoining Agent'"/> <assign name="state" expr="'UnjoiningAgent'"/> <unjoin id1="conferenceid" id2="AgentConnectionid"/> </transition>`
2	Remove the user from the music on hold conference.	`<unjoin id1="onHoldConferenceid" id2="UserConnectionid"/>`
3	Join the user to the call recording conference.	`<join id1="conferenceid" id2="UserConnectionid" entertone="false" exittone="false">/`

Task 7: Responding to different disconnect scenarios

The following examples show how the application responds to a disconnect request when:

- A user is joined to an agent
- A user is joined to a supervisor
- An agent is joined to a supervisor

An example of a general disconnect handler is also presented.

Disconnecting when a user is joined to an agent

The application uses the following logic when a user is joined to an agent in the call recording conference, and the user, agent, or supervisor disconnects from the conference:

If this party disconnects...	The CCXML application...
User	Ends, because the conversation is finished.
Agent	Ends, because the conversation is finished.
Supervisor	Logs the fact that the supervisor disconnected from the conference, and changes state.

The application implements this scenario with the following code:

```
<transition state="RecordingUserAgent" event="connection.disconnected" name="evt">
  <if cond="evt.connectionid == UserConnectionid">
    <log label="'+++Third Party Call Use Case+++'" expr="'User disconnected. Exiting.'"/>

    <exit/>
  <elseif cond="evt.connectionid == SupervisorConnectionid"/>
    <log label="'+++Third Party Call Use Case+++'" expr="'Supervisor Disconnected.'"/>
  <elseif cond="evt.connectionid == AgentConnectionid"/>
    <log label="'+++Third Party Call Use Case+++'" expr="'Agent Disconnected.'"/>
    <exit/>
  </if>
</transition>
```

Disconnecting when a user is joined to a supervisor

The application uses the following logic when a user is joined to a supervisor, and the user, agent, or supervisor disconnects from the call recording conference:

If this party disconnects...	The CCXML application...
User	Ends, because the conversation is finished.
Agent	Logs the fact that the agent disconnected from the conference, and changes state.
Supervisor	Ends, because the conversation is finished.

The application implements this scenario with the following code:

```
<transition state="RecordingUserSupervisor" event="connection.disconnected"
  name="evt">
  <if cond="evt.connectionid == UserConnectionid">
    <log label="'+++Third Party Call Use Case+++'" expr="'User disconnected. Exiting.'"/>
     exit/> <!-- Must notify AppServer that the User disconnected -->
  <elseif cond="evt.connectionid == SupervisorConnectionid"/>

  <log label="'+++Third Party Call Use Case+++'" expr="'Supervisor Disconnected. Joining
      User  to Agent.'"/>
    <assign name="state" expr="'UnjoiningSupervisor'"/>
    <send target="session.id" targettype="'ccxml'" data="'conference.unjoined'"/>
  <elseif cond="evt.connectionid == AgentConnectionid"/>
    <log label="'+++Third Party Call Use Case+++'" expr="'Agent Disconnected. Joining
      User to Supervisor'"/>
    <assign name="state" expr="'UnjoiningAgent'"/>
    <send target="session.id" targettype="'ccxml'" data="'conference.unjoined'"/>
  </if>
</transition>
```

Disconnecting when an agent is joined to a supervisor

The application uses the following logic when an agent is joined to a supervisor, and the user, agent, or supervisor disconnects from the call recording conference:

If this party disconnects...	The CCXML application...
User	Ends, because the conversation is finished.
Agent	Logs the fact that the agent disconnected from the conference, and changes state.
Supervisor	Logs the fact that the supervisor disconnected from the conference, and changes state.

The application implements this scenario with the following code:

```
<transition state="RecordingUserAgentSupervisor" event="connection.disconnected" name="ev
t">
  <if cond="evt.connectionid == UserConnectionid">

<log label="'+++Third Party Call Use Case+++'" expr="'User disconnected. Exiting.'"/>
  <exit/>
  <elseif cond="evt.connectionid == SupervisorConnectionid"/>
     <log label="'+++Third Party Call Use Case+++'" expr="'Supervisor Disconnected.'"/>
     <assign name="state" expr="'RecordingUserAgent'"/>
  <elseif cond="evt.connectionid == AgentConnectionid"/>
     <log label="'+++Third Party Call Use Case+++'" expr="'Agent Disconnected.'"/>
     <assign name="state" expr="'RecordingUserSupervisor'"/>
  </if>
</transition>
```

General disconnect handler

The application contains a general disconnect handler that is triggered when a user, agent, or supervisor disconnects from the call recording conference, and the previous handlers are not triggered. (In most cases, a disconnect event would trigger one of the previous disconnect handlers.)

The general disconnect handler uses the following logic:

If this party disconnects...	The CCXML application...
User	Ends, because the conversation is finished, and notifies the application server that the user disconnected from the conference.
Agent	Logs the fact that the agent disconnected from the conference, and changes state.
Supervisor	Logs the fact that the agent disconnected from the conference, and changes state.

The application implements this scenario with the following code:

```
<transition event="application.disconnectUser" name="evt">
  <if cond="evt.connectionid == UserConnectionid">

<log label="'+++Third Party Call Use Case+++'" expr="'User disconnected. Exiting.'"/>
  <exit/> <!-- Must notify AppServer that the User disconnected -->
  <elseif cond="evt.connectionid == SupervisorConnectionid"/>

<log label="'+++Third Party Call Use Case+++'" expr="'Supervisor Disconnected. Joining
      User to Agent.'"/>
    <assign name="state" expr="'UnjoiningSupervisor'"/>
    <send target="session.id" targettype="'ccxml'" data="'conference.unjoined'"/>
  <elseif cond="evt.connectionid == AgentConnectionid"/>
    <log label="'+++Third Party Call Use Case+++'" expr="'Agent Disconnected. Joining
```

```
      User to Supervisor'"/>
    <assign name="state" expr="'UnjoiningAgent'"/>
    <send target="session.id" targettype="'ccxml'" data="'conference.unjoined'"/>
  </if>
<transition event="connection.disconnected" name="evt">
  <if cond="evt.connectionid == UserConnectionid">

<log label="'+++Third Party Call Use Case+++'" expr="'User disconnected. Exiting.'"/>
  <exit/>
  <elseif cond="evt.connectionid == SupervisorConnectionid"/>
    <log label="'+++Third Party Call Use Case+++'" expr="'Supervisor Disconnected.'"/>
  <elseif cond="evt.connectionid == AgentConnectionid"/>
    <log label="'+++Third Party Call Use Case+++'" expr="'Agent Disconnected.'"/>
  </if>
</transition>
```

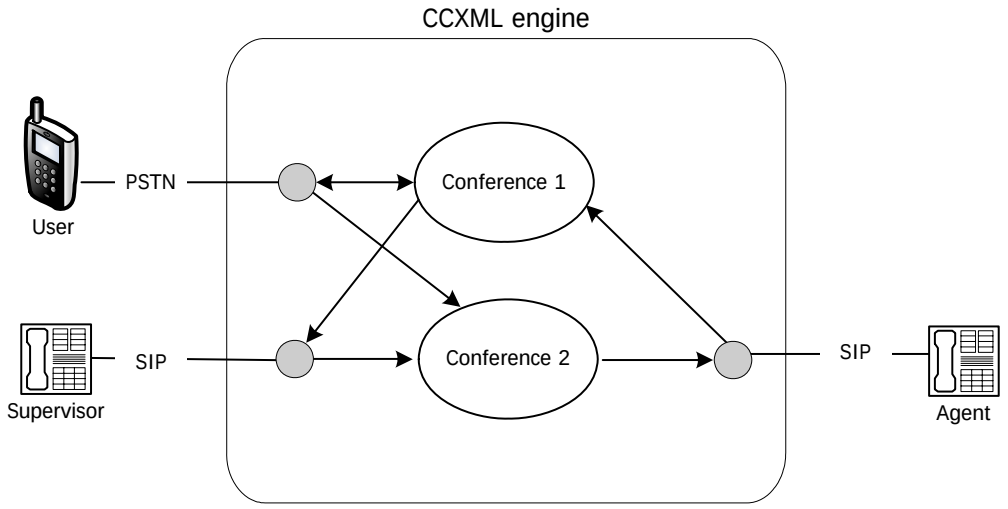
Whisper coaching

The whisper coaching application enables a supervisor to "whisper" advice to an agent without being heard by the user. In this scenario, the application sets up a two-way conversation between the agent and the user, a two-way conversation between the supervisor and the agent, and a listen-only conversation between the supervisor and the user.

This application uses the *www/Coaching/incoming.ccxml* file in the *vx/demos/ccxml-demos* directory. This application does the following:

- Creates two conferences.
- Accepts incoming user calls.
- Joins an incoming user call to conference 1 with a full duplex connection.
- Joins an incoming user call to conference 2 with a half duplex connection.
- Joins the agent to both conferences with half duplex connections.
- Joins a supervisor to both conferences with half duplex connections.

The following diagram illustrates the connections that the whisper coaching application creates:



Resources and capacity

The whisper coaching sample application uses three DS0 resources and six conferencing resources when the incoming user call is a SIP call, and two DS0 resources and six conferencing resources when the incoming user call is a PSTN call. The following table shows the tasks for which these resources are used:

Task	Resources needed
Accepting an incoming user call.	1 DS0 (for a SIP call only)
Contacting the agent.	1 DS0
Contacting the supervisor.	1 DS0
Creating two three-party conferences that consists of the user, agent, and call recording dialog.	6 conferencing

Each media board supports a maximum of 448 conferencing resources. Therefore, in this scenario, the port limitation comes from the conferencing resources instead of the DS0 resources, because six conference resources are required per available port.

Running the application

The following table describes how to run the whisper coaching application. These instructions assume that you already copied the contents of *vx/demos/ccxml-demos/www* to *apache/htdocs*, and *vx/demos/ccxml-demos/cgi-bin to apache/cgi-bin*, as described in *Overview of CCXML sample applications* on page 23.

Step	Description

1	Modify the vx/*demos/ccxml-demos/Coaching/incoming.ccxml* file as follows:
	• Change ***sip:agent*** to specify the SIP URI for the agent:
	`<createcall dest="'sip:agent'">`
	• Change ***sip:supervisor*** to specify the SIP URI for the agent:
	`<createcall dest="'sip:supervisor'">`

2	In the Vision™ Console, define the whisper coaching application to the Vision Call Server by associating a phone number with the following URL: : `http://webserver/Coaching/incoming.ccxml` where ***webserver*** is the URL of the web server that serves the *vx/demos/ccxml-demos/www/Coachingl/incoming.ccxml* file. For information about defining CCXML applications using the Vision™ Console, see the *Dialogic® Vision™ CX Video Gateway Administration Manual* or the *Dialogic® Vision™ VX Integrated Media Platform User's Manual*. For information about defining CCXML applications directly in the application definition file, see *CCXML application definition file* on page 19.
3	Make a call to the number you specified in Step 2.
4	After the scripts run, ensure that the audio was played back to the audio terminal.

Application logic

The whisper coaching application follows this logic:

Step	Action	Code snippet
1	Create a three-party conference with ID conferenceid1.	`<createconference conferenceid="conferenceid1" confname="'conf1'" reservedtalkers="3" reservedlisteners="0" />`
2	Create a second three-party conference with ID conferenceid2.	`<createconference conferenceid="conferenceid2" confname="'conf2'" reservedtalkers="3" reservedlisteners="0" />`
3	Accept an incoming user call.	`<accept connectionid="UserConnectionid" />`
4	Place a SIP call to an agent.	`<createcall dest="'sip:agent'"/>`
5	Join the user to conferenceid1 with a full-duplex connection. The user and agent can have a two-way conversation through this connection.	`<join id1="conferenceid1" id2="UserConnectionid" entertone="false" exittone="false"/>`
6	Join the user to conferenceid2 with a half-duplex connection. The agent and supervisor can hear the user through this connection.	`<join id1="conferenceid2" id2="UserConnectionid" duplex="'half'" entertone="false" exittone="false"/>`

Step	Action	Code snippet
7	Join the agent to conferenceid1 using a half-duplex connection. The agent can talk to the user and supervisor through this connection.	`<join id1="conferenceid1" id2="AgentConnectionid" duplex="'half'" entertone="false" exittone="false"/>`
8	Join the agent to conferenceid2 using a half-duplex connection. The agent can hear the user and supervisor through this connection.	`join id1="AgentConnectionid" id2="conferenceid2" duplex="'half'" entertone="false" exittone="false"/>`
9	Place a SIP call to a supervisor.	`<createcall dest="'sip:supervisor'"/>`
10	Join the supervisor to confereceid1 using a half-duplex connection. The supervisor can hear the user and agent through this connection.	`<join id1="SupervisorConnectionid" id2="conferenceid1" duplex="'half'" entertone="false" exittone="false"/>`
11	Join the supervisor to conferenceid2 using a half-duplex connection. The supervisor an talk to the agent through this connection.	`<join id1="conferenceid2" id2="SupervisorConnectionid" duplex="'half'" entertone="false" exittone="false"/>`
12	Upon receipt of the connection.disconnected event, check which party disconnected. If the user disconnected, then end the session.	`<transition event="connection.disconnected" name="evt"> <if cond="evt.connectionid == UserConnectionid"> <log label="'+++Coaching Use Case+++'" expr="'User disconnected. Exiting.'"/> <exit/> <elseif cond="evt.connectionid == SupervisorConnectionid"/> <log label="'+++Coaching Use Case+++'" expr="'Supervisor Disconnected.'"/> <elseif cond="evt.connectionid == AgentConnectionid"/> <log label="'+++Coaching Use Case+++'" expr="'Agent Disconnected.'"/> </if> <exit/> </transition>`

5 Gateway routing table application

Overview of the Gateway routing table application

The gateway application is a CCXML script called *gateway.ccxml,* located in the *vx\callserver\www\ccxml* directory, that routes PSTN or SIP calls based on information in the gateway routing table. This information includes:

- Valid PSTN numbers or URIs for the called party and calling party on the incoming side of the gateway.
- Valid PSTN numbers or URIs for the called party and calling party on the outgoing side of the gateway.
- A call mode that indicates whether the call is a direct call or a transfer.

Users enter information in the gateway routing table using the Vision™ Console, as described in the *Dialogic® Vision™ CX Video Gateway Administration Manual*.

The gateway routing table provides adequate routing capabilities for most purposes. However, if you need routing functionality that goes beyond what the gateway routing table can provide, you can modify *gateway.ccxml*.

Note: Before modifying *gateway.ccxml,* be sure you are familiar with how the gateway routing table works and with CCXML Version 1.0, which is based upon the W3C Working Draft of CCXML dated 29 June 2005. For more information, see the *Dialogic® Vision™ CX Video Gateway Administration Manual* and http://www.w3.org/TR/2005/WD-ccxml-20050629.

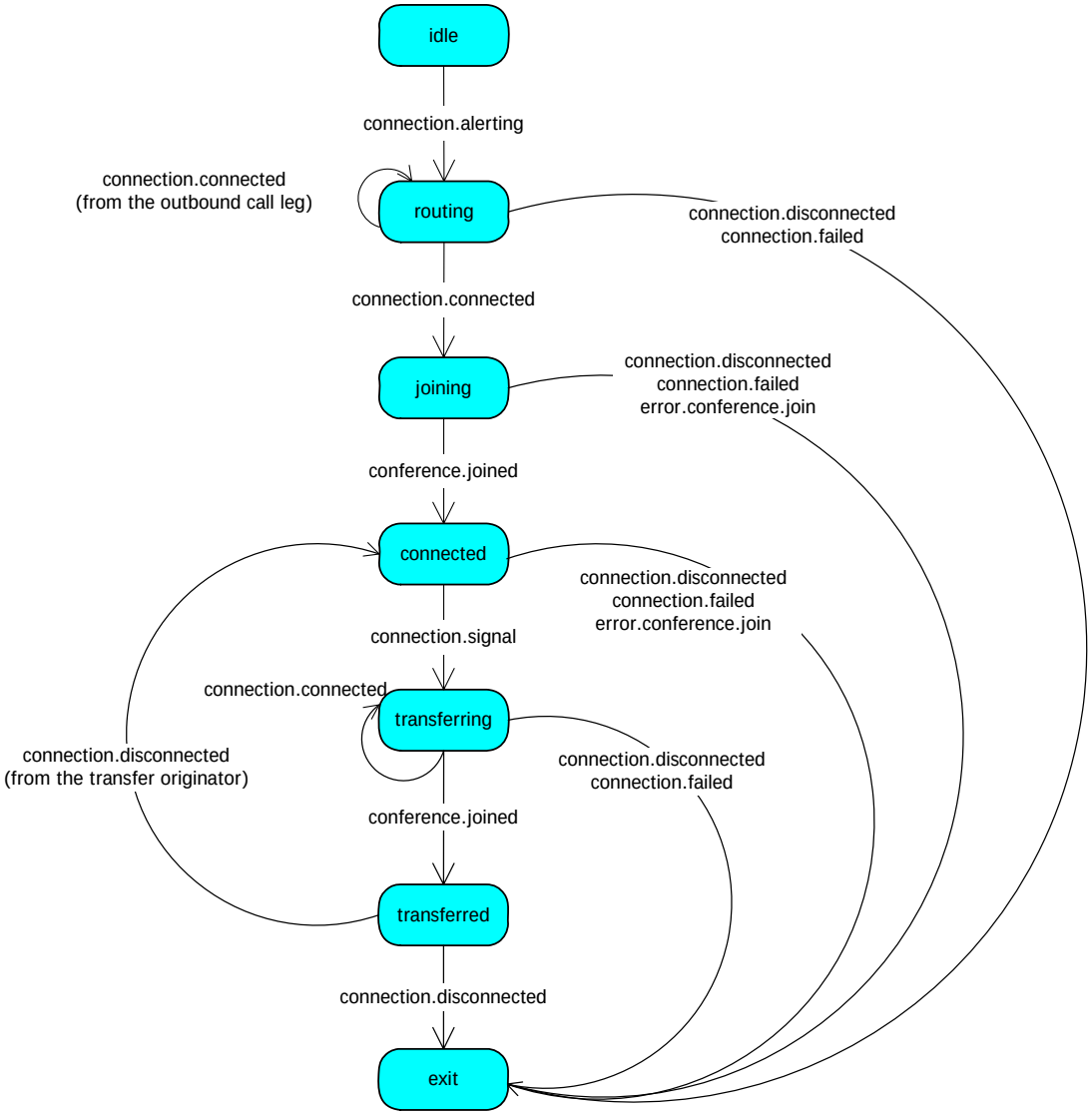
Object model

gateway.ccxml uses the connection objects from the CCXML specification. It also uses the proprietary info object, which contains the following properties:

Property	Description
fromregex	Regular expression that matches the *to* URI on the inbound side of the gateway.
fromstr	Regular expression that matches the *from* URI on the outbound side of the gateway.
tohints	Hints parameter to use when creating the outbound call.
toregex	Regular expression that matches the *to* URI on the inbound side of the gateway.
tostr	Regular expression that matches the *to* URI on the outbound side of the gateway.

Call states

The following illustration shows the call states present in *gateway.ccxml* and the connections between them:



The following table describes the call states for *gateway.ccxml*:

Call state	Description
idle	Initial call state.
routing	A call reaches the routing state when the gateway receives a connection.alerting event, indicating that a new call is present. During this state, the gateway: 1. Uses the Javascript **convertURI** function to convert the *to and from* inbound URIs to the *to and from* outbound URIs, based on the information in the gateway routing table. 2. Creates the outbound call leg.
joining	A call reaches the joining state when the gateway receives a connection.connected event for the outbound call leg, indicating that the outbound call leg is connected to the gateway. During this state, the gateway accepts the inbound call and joins the inbound and outbound call legs together.
connected	A call reaches the connected state when the gateway receives a conference.joined event, indicating that the inbound and outbound call legs are joined together in a call. During this state, the gateway can receive a connection.signal event, indicating a transfer request. In this case, the application creates the new outbound call leg.
transferring	A call reaches the transferring state when the gateway receives a connection.signal event from one of the SIP legs.
transferred	A call reaches the transferred state when the gateway receives a conference.joined event, indicating that the inbound and outbound call legs are joined together in a call.
exit	A call reaches the exit state when the gateway receives a: • connection.disconnected event, indicating that a call leg disconnected. • connection.failed or error.conference.join event, indicating an error during call processing.

Routing a direct call

The following table describes how the Gateway routing table application routes a direct (non-transferred) call. For information about disconnect and error handling, see *Disconnect and error handling* on page 55.

Step	Action	Code snippet

1	The call state is initially set to idle.
	Upon receiving a connection.alerting event, assign values to the info object properties based on the connection.alerting event fields.
		`<!-- ********************************************`
		`# "INIT" :  ALERTING`
		`********************************************** -->`
		`<transition state="idle" event="connection.alerting" name="evt">`
		`<log label="vxglbl" expr="'Session started : alerting received`
		`(' + evt.connectionid + ') protocol(' + evt.protocol + ')`
		`local(' + evt.connection.local + ') remote`
		`(' + evt.connection.remote + ')'"/>`
		`<log label="vxglbl" expr="'Routing entry : To incoming`
		`(' + evt.info.toregex + ')`
		`From incoming (' + evt.info.fromregex + ')`
		`To Outgoing (' + evt.info.tostr + ')`
		`From Outgoing (' + evt.info.fromstr + ')`
		`To Hints (' + evt.info.tohints + ')'"/>`
		`<assign name="in_connectionid" expr="evt.connectionid"/>`
		`<assign name="in_callingparty" expr="evt.connection.remote"/>`
		`<assign name="in_calledparty" expr="evt.connection.local"/>`

2	If the replace string for the *to outbound* URI is	`<if cond="evt.info.tostr == 'reject'">`
		`<log label="vxglbl" expr="'Rejecting inbound call.'"/>`
		`<reject />`

Step	Action	Code snippet
	set to reject, then reject the inbound call. The call state changes to exit. Otherwise, create and call the *to* URI on the outbound side of the gateway. The call state changes to routing.	<!-- Here we couldn't care less about the result of the reject --> <exit /> <else /> <assign name="out_calledparty" expr="convertUri(evt.connection.local, evt.info.toregex, evt.info.tostr) + ';callmode=' + evt.connection.callmode" /> <var name="callerid" expr="convertUri(in_callingparty, evt.info.fromregex, evt.info.fromstr) <log label="vxglbl" expr="'Creating outbound call to the following destination URI : ' + out_calledparty + '. Caller ID: ' + callerid"/> <var name="hints" expr="'{delayack:\'true\'}'" /> <if cond="evt.info.tohints != ''"> <assign name="hints" expr="hints + ',' + evt.info.tohints" /> </if> <assign name="hints" expr="hints + '}'" /> <log label="vxglbl" expr="'Creating call with the following hints (' + hints + ')'" /> <createcall connectionid="out_connectionid" dest="out_calledparty" callerid="callerid" hints="evaluateObject(hints)"/> <assign name="state" expr="'routing'" /> </if> </transition>
3	Accept the inbound call, and join it to the outbound call. The call state changes to joining.	<!-- ** # "ROUTING" : CONNECTED ** --> <transition state="routing" event="connection.connected" name="evt"> <if cond="evt.connectionid == out_connectionid"> <log label="vxglbl" expr="'Outbound leg answered. Accepting inbound call.'" /> <accept connectionid="in_connectionid" /> <elseif cond="evt.connectionid == in_connectionid" /> <log label="vxglbl" expr="'Inbound call accepted. Joining legs.'" /> <join id1="in_connectionid" id2="out_connectionid" /> <assign name="state" expr="'joining'"/> </if> </transition>
4	Upon receipt of the conference.join event, the call legs are successfully joined. The call state changes to connected.	<!-- ** # "JOINING" : JOINED ** --> <transition state="joining" event="conference.joined" name="evt"> <log label="vxglbl" expr="'Gateway call established.'" /> <assign name="state" expr="'connected'"/> </transition>

Routing a transferred call

The following table describes how the Gateway routing table application routes a transferred call successfully. For information about disconnect and error handling, see *Disconnect and error handling* on page 55.

Step	Action	Code snippet

1	The call state is initially set to idle.
	Upon receiving a connection.alerting event, assign values to the info object properties based on the connection.alerting event fields.
		`<!-- ********************************************`
		`# "INIT" :  ALERTING`
		`********************************************** -->`
		`<transition state="idle" event="connection.alerting" name="evt">`
		`<log label="vxglbl" expr="'Session started : alerting received`
		`(' + evt.connectionid + ') protocol(' + evt.protocol + ')`
		`<local(' + evt.connection.local + ') remote`
		`(' + evt.connection.remote + ')'"/>`
		`<log label="vxglbl" expr="'Routing entry : To incoming`
		`(' + evt.info.toregex + ')`
		`From incoming (' + evt.info.fromregex + ')`
		`To Outgoing (' + evt.info.tostr + ')`
		`From Outgoing (' + evt.info.fromstr + ')`
		`To Hints (' + evt.info.tohints + ')'"/>`
		`<assign name="in_connectionid" expr="evt.connectionid"/>`
		`<assign name="in_callingparty" expr="evt.connection.remote"/>`
		`<assign name="in_calledparty" expr="evt.connection.local"/>`

2	If the replace string for the *to outbound* URI is set to reject, then reject the inbound call. The call state changes to Exit.
	Otherwise, create and call the *to* URI on the outbound side of the gateway. The call state changes to routing.
		`<if cond="evt.info.tostr == 'reject'">`
		`<log label="vxglbl" expr="'Rejecting inbound call.'"/>`
		`<reject />`
		`<!-- Here we couldn't care less about the result of`
		`the reject -->`
		`<exit />`
		`<else />`
		`<assign name="out_calledparty"`
		`expr="convertUri(evt.connection.local, evt.info.toregex,`
		`evt.info.tostr) + ';callmode=' +`
		`evt.connection.callmode" />`
		`<var name="callerid"`
		`expr="convertUri(in_callingparty, evt.info.fromregex,`
		`evt.info.fromstr)`
		
		`<log label="vxglbl" expr="'Creating outbound call to the following`
		`destination URI : ' + out_calledparty + '. Caller ID: '`
		`+ callerid"/>`
		
		`<var name="hints" expr="'{delayack:\'true\''" />`
		`<if cond="evt.info.tohints != ''">`
		`<assign name="hints" expr="hints + ',' + evt.info.tohints" />`
		`</if>`
		`<assign name="hints" expr="hints + '}'" />`
		`<log label="vxglbl" expr="'Creating call with the following hints`
		`(' + hints + ')'" />`
		
		`<createcall connectionid="out_connectionid"`
		`dest="out_calledparty"`
		`callerid="callerid"`
		`hints="evaluateObject(hints)"/>`
		
		`<assign name="state" expr="'routing'" />`
		`</if>`
		`</transition>`

3	Accept the inbound call and join it to the outbound call. The call state changes to joining.	`<!-- ********************************************`
		`# "ROUTING" :  CONNECTED`
		`********************************************** -->`
		`<transition state="routing" event="connection.connected" name="evt">`
		`<if cond="evt.connectionid == out_connectionid">`
		`<log label="vxglbl" expr="'Outbound leg answered. Accepting`
		`inbound call.'" />

Step	Action	Code snippet
		<accept connectionid="in_connectionid" /> <elseif cond="evt.connectionid == in_connectionid" /> <log label="vxglbl" expr="'Inbound call accepted. Joining legs.'" /> <join id1="in_connectionid" id2="out_connectionid" /> <assign name="state" expr="'joining'"/> </if> </transition>
4	Upon receipt of the joining event, the call legs are successfully joined. The call state changes to connected.	<!-- ********** # "JOINING" : JOINED ********** --> <transition state="joining" event="conference.joined" name="evt"> <log label="vxglbl" expr="'Gateway call established.'" /> <assign name="state" expr="'connected'"/> </transition>
5	Upon receipt of a connection.signal event with a substate of TRANSFER, determine which call leg received the transfer. If the replace string for the *to outbound* URI is set to reject, then disconnect the inbound call. The call state changes to exit. Otherwise, create the new *to* URI on the outbound side of the gateway, and connect the inbound call to the new destination. The call state changes to transferring.	<transition state="connected" event="connection.signal" name="evt"> <if cond="evt.connection.substate == 'TRANSFER'"> <log label="vxglbl" expr="'Transfer request received on SIP call leg. URI(' + evt.info.URI + ').'"/> <log label="vxglbl" expr="'Routing entry : To incoming (' + evt.info.toregex + ') From incoming (' + evt.info.fromregex + ') To Outgoing (' + evt.info.tostr + ') From Outgoing (' + evt.info.fromstr + ') To Hints (' + evt.info.tohints + ')'"/> <assign name="transfer_connectionid" expr="evt.connectionid" /> <if cond="transfer_connectionid == out_connectionid"> <log label="vxglbl" expr="'Transfer request received on outbound leg.'" /> <elseif cond="transfer_connectionid == in_connectionid" /> <log label="vxglbl" expr="'Transfer request received on inbound leg.'" /> <!-- In the tranferring state, we assume the in_connectionid references the call that is still active on the system. --> <assign name="in_connectionid" expr="out_connectionid" /> <assign name="in_callingparty" expr="out_calledparty" /> </if> <if cond="evt.info.tostr == 'reject'"> <log label="vxglbl" expr="'Transfer request rejected. Disconnecting inbound call.'"/> <disconnect connectionid="in_connectionid"/> <!-- Here we couldn't care less about the result of the disconnect --> <exit /> <else /> <assign name="out_calledparty" expr="convertUri(evt.info.URI, evt.info.toregex, evt.info.tostr)" /> <!-- Caller id taken from the call remaining on the platform. --> <var name="callerid" expr="convertUri(in_callingparty, evt.info.fromregex, evt.info.fromstr)"/> <log label="vxglbl" expr="'Transfering to (' + out_calledparty + ') from (' + callerid + ').'" /> <!-- Call already unjoined by the platform so we can

Step	Action	Code snippet

		proceed with the outbound call.
		-->`<var name="hints" expr="'{'" />`
		`<if cond="evt.info.tohints != ''">`
		`<assign name="hints"`
		`expr="hints + evt.info.tohints" />`
		`</if>`
		`<assign name="hints" expr="hints + '}'" />`
		`<log label="vxglbl" expr="'Creating call with the`
		`following hints`
		`(' + hints + ')'" />`
		`<createcall joinid="in_connectionid"`
		`connectionid="out_connectionid"`
		`dest="out_calledparty"`
		`callerid="callerid"`
		`hints="evaluateObject(hints)"`
		`/>`
		`<assign name="state" expr="'transferring'"/>`
		`</if>`
		`</if>`
		</transition>

6	Upon receipt of the connection.connected event, change the call state to transferred.
	Upon receipt of the conference.joined event (indicating the call legs are successfully joined), disconnect the call leg that initiated the transfer and change the call state to transferred.
		<!-- **********
		# "TRANSFERRING" : CONNECTED
		********** -->`<transition state="transferring" event="connection.connected"`
		`name="evt">`
		<log label="vxglbl" expr="'Outbound leg answered. Joining legs.'" />
		</transition>
		
		<!-- **********
		# "TRANSFERRING" : JOINED
		********** -->`<transition state="transferring" event="conference.joined"`
		`name="evt">`
		<log label="vxglbl" expr="'Legs joined. Transfer completed.
		Disconnecting call leg that initiated the transfer.'" />
		<if cond="transfer_connectionid == -1">
		`<assign name="state" expr="'connected'"/>`
		<else />
		
		<log label="vxglbl" expr="'Disconnecting SIP transfer originating
		call leg.'" />
		`<assign name="state" expr="'transferred'"/>`
		`<disconnect connectionid="transfer_connectionid" />`
		</if>
		</transition>

Disconnect and error handling

If the gateway receives a disconnect event or error notification in any of the call states except initial and exit, the call state changes to exit. The following table describes how the gateway routing table application handles disconnects and errors:

Scenario / Description	Code snippet

Scenario: Participant in a direct or transferred call hangs up while the call is in the routing, joining, or connected states.
Description: Upon receipt of a
	`<transition state="`***gw_state***`" event="connection.disconnected"`
	`name="evt">`
	<if cond="evt.connectionid == out_connectionid">
	`<log label="vxglbl" expr="'Outbound leg disconnected.`
	`Disconnecting inbound leg.'" />`
	`<reject connectionid="in_connectionid" />`
	<elseif cond="evt.connectionid == in_connectionid" />
	`<log label="vxglbl" expr="'Inbound leg disconnected.`
	`Disconnecting outbound leg.'" />`
	`<disconnect connectionid="out_connectionid" />`

Scenario / Description	Code snippet

connection.disconnected event during the routing, joining, or connected states, the application disconnects the inbound and outbound call legs.	`</if>`
	`<exit />`
	`</transition>`
	where ***gw_state*** is routing, joining, or connected.

Scenario: Participant in a transferred call hangs up while the call is in the transferring state.
Description: Upon receipt of the connection.disconnected event during the transferring state (indicating one call party hung up), determine which call leg disconnected.
If the transfer originator disconnected, disregard the event.
If the remaining call leg or transfer destination disconnected, disconnect both call legs.
	`<!-- ***********************************************`
	`# "TRANSFERRING" :  DISCONNECTED`
	`*********************************************** -->`
	`<transition state="transferring" event="connection.disconnected"`
	`  name="evt">`
	`<if cond="evt.connectionid == transfer_connectionid" >`
	`  <log label="vxglbl" expr="'Referred call leg`
	`  disconnected.'" />`
	`  <assign name="transfer_connectionid" expr="-1" />`
	`<elseif cond="evt.connectionid == out_connectionid" />`
	`  <log label="vxglbl" expr="'Outbound leg disconnected.`
	`  Disconnecting inbound leg.'" />`
	`  <disconnect connectionid="in_connectionid" />`
	`  <disconnect connectionid="transfer_connectionid" />`
	`  <exit />`
	`<elseif cond="evt.connectionid == in_connectionid" />`
	`  <log label="vxglbl" expr="'Inbound leg disconnected.`
	`  Disconnecting outbound leg.'" />`
	`  <disconnect connectionid="out_connectionid" />`
	`  <disconnect connectionid="transfer_connectionid" />`
	`  <exit />`
	`</if>`
	`</transition>`

Scenario: Participant in a transferred call hangs up while the call is in the transferred state.
Description: Upon receipt of the connection.disconnected event during the transferred state (indicating one call party hung up), determine which call leg disconnected.
If the transfer originator disconnected, disregard the event.
If the remaining call leg or transfer destination disconnected, disconnect the remaining call leg.
Note: In the transferred state, the transfer originator should already be disconnected.
	`<!-- ***********************************************`
	`# "TRANSFERRED" :  DISCONNECTED`
	`*********************************************** -->`
	`<transition state="transferred" event="connection.disconnected"`
	`  name="evt">`
	`<if cond="evt.connectionid == transfer_connectionid">`
	`  <log label="vxglbl" expr="'Call leg that originated`
	`  the transfer disconnected.'" />`
	`  <assign name="state" expr="'connected'"/>`
	`<elseif cond="evt.connectionid == out_connectionid" />`
	`  <log label="vxglbl" expr="'Outbound leg disconnected.`
	`  Disconnecting inbound leg.'" />`
	`  <disconnect connectionid="in_connectionid" />`
	`  <exit />`
	`<elseif cond="evt.connectionid == in_connectionid" />`
	`  <log label="vxglbl" expr="'Inbound leg disconnected.`
	`  Disconnecting outbound leg.'" />`
	`  <disconnect connectionid="out_connectionid" />`
	`  <exit />`
	`</if>`
	`</transition>`

Scenario: Application cannot create the outbound call leg in the	`<transition state="routing" event="connection.failed"`
	`  name="evt">`
	`<if cond="evt.connectionid == out_connectionid">`

Scenario / Description	Code snippet
routing state. **Description:** Upon receipt of a connection.failed event during the routing state, the application returns an error message and disconnects the inbound call leg.	`<log label="vxglbl" expr="'Outbound leg failed with reason ' + evt.reason +'. Rejecting inbound leg.'" /> <reject connectionid="in_connectionid" reason="'486 User busy'" /> <elseif cond="evt.connectionid == in_connectionid" /> <log label="vxglbl" expr="'Inbound leg failed with reason ' + evt.reason +'. Disconnecting outbound leg.'" /> <disconnect connectionid="out_connectionid" /> </if> <exit /> </transition>`
Scenario: Application cannot join the inbound and outbound call legs in the joining state. **Description:** Upon receipt of an error.conference.join event during the joining state, the application returns an error message disconnects the inbound and outbound legs.	`<transition state="routing" event="error.conference.join" name="evt"> <log label="vxglbl" expr="'Unable to join both call legs.'" /> <reject connectionid="in_connectionid" /> <disconnect connectionid="out_connectionid" /> <exit /> </transition>`
Scenario: Application cannot create the outbound call leg in the transferring state. **Description:** Upon receipt of a connection.failed event during the transferring state, the application tests to see whether the failure occurred in the inbound or outbound call leg. Then it displays an appropriate error message and disconnects the call leg with the error. It also disconnects the transfer originator at the end of the transition.	`<transition state="transferring" event="connection.failed". name="evt"> <if cond="evt.connectionid == out_connectionid"> <log label="vxglbl" expr="'Outbound leg failed with reason ' + evt.reason +'. Disconnecting inbound leg.'" /> <disconnect connectionid="in_connectionid" /> <elseif cond="evt.connectionid == in_connectionid" /> <log label="vxglbl" expr="'Inbound leg failed with reason ' + evt.reason +'. Disconnecting outbound leg.'" /> <disconnect connectionid="out_connectionid" /> </if> <disconnect connectionid="transfer_connectionid" /> <exit /> </transition>`

6 Glossary

A

ADTCP: An audio driver that provides a TCP interface to MIOSIP for rendering SSML fragments.

AMR: Adaptive multi-rate; an audio data compression scheme optimized for speech coding. This scheme was adopted by 3GPP and is used in video services.

ASR: Automatic speech recognition; ASR resources, called ASR engines in the MRCP framework, typically enable users of information systems to speak entries rather than punching numbers on a keypad. See also MRCP.

Authorization and Usage Indication interface: XML-over-HTTP mechanism that authorizes call sessions and gathers information for call detail reports.

B

blind transfer: A call transfer in which the originating caller is not announced and is connected directly to destination. In a blind transfer the Vision™ Server redirects the caller to the callee without remaining in the connection and does not monitor the outcome.

bridge transfer: A blind transfer in which the Vision™ Server redirects the caller to the callee and remains as a listener.

C

Call Server: Component of the Vision™ Server that manages call control and routing capabilities.

CallPlacer interface: XML-over-HTTP mechanism for initiating outbound sessions or calls for VoiceXML applications.

CCXML: Call Control Extensible Markup Language; a W3C Working Draft standard language for providing telephony call control support for dialog systems, gateways, and conferencing services.

CCXML application definition file: A file that maps individual CCXML applications to number ranges that trigger the execution of those applications.

clock: A periodic reference signal used for synchronization on a transmission facility, such as a telephony bus. See also clock master, clock slave, clock fallback.

clock master: A board that drives the clock signal for a system of boards connected by a bus cable. See also clock slave.

clock slave: A board that derives its clock signal from a bus cable; the clock signal is driven by the bus clock master. See also clock master.

consultation transfer: A call transfer in which the Vision™ Server initiates a transfer between two parties, but does not stay attached to the call once it is

successfully established. The caller remains connected to the Vision™ Server if the transfer fails.

D

DTMF: Dual tone multi frequency; an inband signaling system that uses two simultaneous voiceband tones for dialing. Also called touchtone. Some times DMTF is used to generally describe any telephony keypad press, even if tones are not generated.

G

G.711: An ITU PCM encoder/decoder specification for mu-law and A-law encoding.

H

H.100 bus: A TDM telephony bus standard for integrating hardware from various PC board vendors. The H.100 specification defines a ribbon cable bus that transports telephony voice data and signaling data across PCI boards. The H.100 bus is an interoperable superset of the H-MVIP and MVIP-90 telephony buses.

H.223: A protocol used to multiplex control and audio and video media on and off of a single DS0 within a trunk.

H.263: An ITU video compression standard. H.263 supports CIF, QCIF, SQCIF, 4CIF and 16CIF resolutions.

H.264: An ITU and ISO video compression standard that compresses video into lower bandwidth compared to H.263 and MPEG-4. H.264 is also called MPEG-4 Part 10.

I

INAP: Intelligent Network Application Part; an SS7 protocol that facilitates building platform-independent, transport-independent, and vendor-independent applications. Such applications include service switching points (SSPs), internet protocol (IP) applications, service control points (SCPs), enhanced services platforms, service circuit nodes, and other custom applications.

ISDN: Integrated services digital network; a standard for providing voice and data telephone service with all digital transmission and message-based signaling.

ISUP: ISDN user part; the SS7 protocol layer that allows for the establishment, supervision, and clearing of circuit-switched connections between two SS7 signaling points, such as central office switches. Despite its name, the ISUP layer is not unique to interconnecting. It is used to manage all types of circuit-switched connections.

ITU: International Telecommunications Union; an international standards body for telecommunications.

IVR: Interactive voice response; a telephony application in which callers interact with programs using recorded or synthesized voice prompts, DTMF digits, or speech recognition to query or deliver information.

M

Media Resource Function: Component of the Dialogic® Vision™ VX Integrated Media Platform that provides media processing including record, playback, and interfaces to speech recognition resources. The Media Resource Function is implemented by MIOSIP.

MIB: Management information base; an SNMP collection of objects that represent a managed node. Physically, a list of variables. Logically, a table with rows of variables.

MIOSIP: Implements the Media Resource Function of the Dialogic® Vision™ VX Integrated Media Platform. MIOSIP provides SIP call control, media processing over RTP, DTMF generation and recognition, and an MRCP client to automatic speech recognition (ASR) resources.

MPEG-4: An ISO/IEC standard for compressing multimedia data (video, audio, and speech).

MRCP: Media Resource Control Protocol; an application protocol for implementing automatic speech recognition (ASR) and text-to-speech services (TTS). MRCP provides a distributed system of ASR and TTS engines connected over an IP network.

MTP: Message transfer part; the SS7 protocol layers responsible for the reliable, in-sequence delivery of packets between two SS7 signaling points. The MTP functions include message routing, signaling link management, signaling route management, and congestion control.

MVIP-95: Device driver specification for H-MVIP, H.100, and H.110 telephony buses.

N

NETANN: Basic Network Media Services with SIP; an interface that enables applications in a SIP network to locate and invoke basic services on a media server. These services include network announcements, user interaction, and conferencing services. Also called RFC 4240.

O

OSP: Open Settlement Protocol; a European Telecommunications Standards Institute (ESTI) protocol used to exchange authorization, accounting, and usage information for IP telephony.

P

PSTN: Public switched telephone network; a public telephone network.

R

route: A connection path. On the PSTN network, a route is a logical collection of trunks. On the IP network, a route is a destination URL.

RTP: Real time transport protocol; a layer added to the internet protocol (IP) that addressed problems caused when real-time interactive exchanges (such as

audio data) are conducted over lines designed to carry packet-switched (connectionless) data.

S

SCCP: Signaling connection control part; an SS7 protocol that provides both connection-oriented and connectionless data transfer over an SS7 network. It extends the service provided by the SS7 MTP layers by adding extended addressing capabilities and multiple classes of service. The SCCP addressing capabilities allow a message to be addressed to an individual application or database within a signaling point. See also SS7.

SDP: Session description protocol, a protocol that defines a text-based format for describing streaming media sessions and multicast transmissions.

Signaling Server: An optional component of the Vision™ Server that provides redundant and scalable ISUP signaling.

SIP: Session initiation protocol. An IP signaling and telephony control protocol used mainly for voice over IP calls and multimedia communications. SIP relies on the session description protocol (SDP) for session description and the Real Time Transport Protocol (RTP) for actual transport.

SRGS: Speech Recognition Grammar Specification (SRGS); a syntax for representing the grammars used in speech recognition.

SS7: Signaling system 7; an out-of-band signaling system that provides fast call setup using circuit-switched connections and transaction capabilities for remote database interactions.

SSML: Speech Synthesis Markup Language; a proposed standard for enabling access to the internet using speech. SSML provides a standard way to control various aspects of speech (such as pronunciation, volume, pitch, and rate) over a variety of platforms.

SSML Processor: Component of the Dialogic® Vision™ VX Integrated Media Platform that processes SSML requests for audio and text-to-speech.

T

T.38 fax: A standard for real-time fax over IP that makes it possible for fax machines from different vendors to talk to each other over IP networks. The T.38 standard defines how to conduct group 3 facsimile transmission between terminals in which a portion of the transmission path between terminals includes (besides the PSTN or ISDN) an IP network such as the internet.

TCAP: Transaction capabilities application part; an SS7 protocol that provides applications with transaction support over the SS7 network. It enables the exchange of non-circuit related data, such as database queries and responses and remote feature invocation requests between SS7 signaling points. The TCAP layer relies on both the MTP and SCCP layers for message addressing and delivery.

TDM: Time division multiplexing; a technique for transmitting a number of separate data, voice, or video signals simultaneously over one communications medium by quickly interleaving a piece of each signal one after another.

telecom configuration file: File that provides information about the resources that interface with the Call Server and about other elements, such as the number of routes and the circuit selection.

trunk: The physical interface between the telephone network and the Vision™ Server. In telephone networks, a trunk is a shared connection between two switches. It differs from a line in that it is not dedicated to one subscriber or extension. T1 and E1 trunks carry 24 and 31 circuits, respectively.

TTS: Text-to-speech; a system that converts written language to speech.

V

Vision™ Console: Web-based configuration tool that configures the Vision™ Server.

VoiceXML: Voice Extensible Markup Language; a language that enables users to interact with the internet through voice recognition technology.

VoiceXML application configuration file: A file that maps individual VoiceXML applications to number ranges that trigger the execution of those applications.

VoiceXML Interpreter: Component of the Dialogic® Vision™ VX Integrated Media Platform that interprets VoiceXML dialogs.

VoiceXML Subsystem: Component of the Dialogic® Vision™ VX Integrated Media Platform that provides media processing for VoiceXML applications. The VoiceXML Subsystem consists of the VoiceXML Interpreter, SSML Processor, and Media Resource Function.

Index

A

application definition file 13, 19

audio codecs 19

audio conferencing 16

C

call 7

- disconnecting 55
- error handling 55
- recording 24
- routing 51, 53
- transfering 36

CallPlacer interface 13

CCXML 7

- configuration 7
- elements 7, 8
- session variables 11
- starting a new session 13

ccxmlappcfg.xml file 13, 19

configuration 7

D

data exchange between CCXML and VoiceXML 14

disconnecting a call 55

E

eavesdropping calls 17

elements 7, 8

error handling 55

event processing 14

G

gateway.ccxml 49

L

limitations 7

logging 15

M

media splitting 17

music on hold 27

music on hold with a wait queue 31

P

preferred audio codecs 19

R

resources 24

routing 13, 19, 49, 53

S

sample applications 23

- call recording 24
- music on hold 27
- music on hold with a wait queue 31
- Third party call with transfer out 36
- whisper coaching 45

scenarios 15

session variables 11

standards 7

T

telecom.conf file 13

third party call with transfer out 36

third-party call control 18

transfer scenarios 7

V

VoiceXML 14

W

whisper coaching 45